UNIVERSIDAD AUTONOMA DE MADRID

Escuela Politécnica Superior

18|19

Trabajo fin de grado

Aprendizaje automático aplicado a Sentiment Analysis en castellano

Santiago Palmero Muñoz

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

Campus Internacional excelencia UAM CSIC+

www.uam.es

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería informática

TRABAJO FIN DE GRADO

Aprendizaje automático aplicado a Sentiment Analysis en castellano

Autor: Santiago Palmero Muñoz
Tutor: Jack Mario Mingo Postiglioni
Ponente: Gonzalo Martínez Muñoz

mayo 2019

Santiago Palmero Muñoz
Aprendizaje automático aplicado a Sentiment Analysis en castellano

Santiago Palmero Muñoz

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Abuelo te dedico este trabajo, gracias por tu apoyo y tu interés desde siempre.

Los ordenadores son inútiles. Sólo pueden darte respuestas.

Pablo Picasso

Agradecimientos

Lo primero de todo es agradecerle a mi tutor la posibilidad que me ha dado de poder realizar este proyecto. También quiero agradecerle a mi madre Paloma su ayuda en las correcciones y la revisión de esta memoria, a pesar de no entender absolutamente nada de lo que estaba escrito. No podría faltar mi agradecimiento a Marta por todo el apoyo incondicional que me ha dado durante este tiempo; tiene un valor incalculable. Por último, gracias a mis chicos, Alfredo, Bravo, Cabs, David y Pablo por haber hecho que estos cuatro duros años hayan sido más llevaderos. Aprovecho para recordar que nos queda un viaje pendiente.

Resumen

Sentiment Analysis, vertiente tanto del procesamiento del lenguaje natural como del campo del Aprendizaje Automático, tiene como objetivo clasificar documentos en función de su polaridad, principalmente, positiva y negativa. Este tipo de análisis está resultando en la actualidad de gran utilidad en ámbitos, tanto comerciales como sociales.

En esta memoria se detalla el desarrollo y los resultados obtenidos de la realización de un proyecto de Sentiment Analysis en la lengua española, que cuenta con menor cantidad de proyectos y recursos. La colección empleada contiene cerca de 70.000 mensajes en español de la red social Twitter, divididos en tres clases: positiva, negativa y neutra. Los tweets, pertenecientes a distintas categorías, desde entretenimiento hasta política, presentan estructuras muy diferentes y contienen anglicismos, fallos ortográficos, onomatopeyas, emoticonos, entre otros recursos lingüísticos utilizados en las redes sociales.

Ante esta gran variedad lingüística, se muestra en este informe el beneficio obtenido de la aplicación de distintos tipos de preprocesados sobre los datos originales, en la eficacia de los algoritmos, encargados de la tarea de clasificación. De igual forma, se expone una comparativa de los resultados conseguidos en la clasificación de los tweets, empleando dos técnicas distintas de modelado del lenguaje, sparse vectors y dense vectors, con distintos clasificadores basados en aprendizaje supervisado. El modelado con sparse vectors resulta superior al de dense vectors, salvo cuanto estos vectores se generan de forma supervisada, como se realiza en los modelos de deep learning. Dentro de los clasificadores destaca la regresión logística y los modelos de deep learning: Long Short-Term Memory y Gated Recurrent Unit.

Palabras clave

Análisis de sentimiento, Minería de opinión, Procesamiento del lenguaje natural, Aprendizaje Automático, Extracción de información textual

Abstract

Sentiment Analysis, an area of both Natural Language Processing and the field of Machine Learning, aims to classify documents according to their polarity, mainly positive and negative. This type of analysis is currently proving very useful in both commercial and social spheres.

This report details the development and results obtained from carrying out a Sentiment Analysis project in the Spanish language, of which there are fewer projects and resources. The collection used contains nearly 70,000 messages in Spanish from the social network Twitter, divided into three classes: positive, negative and neutral. The tweets, belonging to different topics, from entertainment to politics, present very different structures and contain anglicisms, misspellings, onomatopoeia, and emoticons, among other linguistic resources used in social networks.

Faced with this great linguistic variety, this report shows the benefit obtained from the application of different types of preprocessing techniques in the effectiveness of the Machine Learning algorithms. In the same way, a comparison of the results obtained in the classification task is exposed, using two different techniques of language modeling, sparse vectors and dense vectors, with different supervised learning algorithms. The modeling with sparse vectors is superior to that of dense vectors, except when these vectors are generated in a supervised manner, as is done in deep learning models. Logistic regression and deep learning models, Long Short-Term Memory and Gated Recurrent Unit, stand out within these algorithms.

Keywords

Sentiment Analysis, Opinion mining, Natural Language Processing, Machine Learning, Information extraction

Índice

Listas

Lista de ecuaciones

Lista de figuras

1

Introducción

Sentiment analysis es una rama del procesamiento del lenguaje natural (denominado en inglés NLP, Natural Language Processing), disciplina que, a su vez, es un área de creciente interés en el campo de la Inteligencia Artificial y del Machine Learning. Su objetivo es intentar substraer de los documentos, que se toman como entrada, información subjetiva, por lo general, connotaciones positivas y negativas. Debido a esta subjetividad, es una tarea de clasificación que no consigue porcentajes de acierto muy elevados. En este TFG, se realizará una clasificación de tweets en castellano que se encuentran etiquetados en tres clases: positivo (P), negativo (N) y neutro (NEU). El hecho de realizar el análisis en castellano resulta un aspecto limitante ya que, como veremos, la cantidad de recursos es mucho menor a la que se puede encontrar en inglés.

Los tweets que contienen las distintas colecciones pueden pertenecer a ámbitos completamente distintos. Esto causa que el contenido de los tweets en cuanto a su estructura sintáctica es muy variable. Además, en las redes sociales se emplea tanto el lenguaje formal como el informal y se usan recursos como onomatopeyas, anglicismos, y emoticonos, entre otros. También, hay un gran número de errores lingüísticos y neologismos. Ante esta variedad, que dificulta la tarea de clasificación, en este trabajo se mostrarán los beneficios de aplicar distintas técnicas de preprocesado sobre los documentos.

Otro aspecto importante es que los clasificadores encargados de decidir a que clase pertenecerá cada documento deben de recibir vectores de entrada de tamaño fijo. Para resolver este problema se emplearán dos estrategias de vectorización, la creación de los denominados sparse vectors y dense vectors.

Por último, se hará una comparación de los resultados para distintos tipos de clasificadores de Machine Learning empleando distintas configuraciones, distintos preprocesados y distintas formas de vectorización.

A continuación, en los siguientes subapartados, se dará información acerca de los objetivos a conseguir y la motivación de este proyecto. Por último, se proporcionará una breve explicación de la organización que seguirá esta memoria.

1.1. Objetivos

Los objetivos que se van a intentar lograr para este proyecto son:

- Explorar las diferencias y limitaciones de realizar sentiment analysis en castellano.
- Analizar la eficacia de distintas técnicas de preprocesado aplicables a los documentos.
- Evaluar la eficacia de las dos técnicas de modelado del lenguaje más populares.
- Contrastar la eficacia de diferentes técnicas de clasificación de texto mediante aprendizaje automático.
- Discutir los resultados y proponer líneas futuras de trabajo.

1.2. Motivación

El procesamiento del lenguaje natural es un campo multidisciplinar en el que se lleva trabajando durante las últimas décadas. La capacidad de utilizar el lenguaje es algo propio de la naturaleza humana e intentar descifrarlo y comprenderlo mediante la computación es verdaderamente de gran importancia. En los últimos años, con la expansión de la web y de las redes sociales, existe una gran cantidad de información a la que se le puede realizar todo tipo de análisis como, por ejemplo, estudios sociológicos a gran escala, análisis de imagen corporativa, o soporte al cliente, entre otros usos. Estos aspectos de gran interés, hacen que sea necesario evaluar las técnicas en las que se basan los mismos y, de esta forma, poder desarrollar una opinión acerca de su efectividad y su recorrido futuro. Igualmente, profundizar más en el procesamiento del lenguaje natural y en general, en sus técnicas empleadas, resulta fundamental para el avance de este campo.

1.3. Organización

Esta memoria se va a dividir en cuatro apartados. En el primero de ellos, *Estado del arte*, se incluirá una explicación sobre aquellos aspectos más destacados del procesado, del modelado y de la clasificación del lenguaje. El segundo apartado, *Diseño y Desarrollo*, proporcionará una explicación detallada de las distintas tareas que se han tenido que llevar a cabo y cómo se han realizado. En el tercer apartado, *Integración, pruebas y resultados*, se aportará información sobre estos aspectos. Finalmente, en el cuarto, *Conclusiones y trabajo futuro*, se expondrán los resultados finales y comentarán aspectos a mejorar o ampliar del proyecto.

2

Estado del arte

Debido a que Sentiment Analysis es una rama del procesamiento del lenguaje natural, denominado en inglés "Natural Language Processing (NLP)", la visión sobre el estado del arte de Sentiment Analysis va a englobar muchos aspectos generales del estado del arte de NLP, ya que van absolutamente ligados. Sin embargo, sí existen ciertos aspectos concretos en los que difieren y en los que se entrará más en detalle. El objetivo de Sentiment Analysis es, a raíz de un documento de entrada, clasificarlo acorde con su polaridad. Esta tarea se suele realizar de forma supervisada, es decir, algoritmos que aprenden de forma automática se entrenan con documentos de los que ya se conoce su polaridad para, posteriormente, poder clasificar documentos de los que se desconoce dicha característica.

La revisión se dividirá en tres subapartados. El primero, considerará el preprocesamiento que puede ser aplicado a estos documentos; el segundo, tratará las dos formas de modelar el contenido de los documentos para que sean clasificados y, el tercero, se referirá a las técnicas de clasificación más utilizadas. No sólo se van a mencionar las técnicas más actuales, sino también en qué contexto pueden ser aplicadas y cuáles son sus pros y sus contras.

Un aspecto a tener en cuenta de forma general en los proyectos de Sentiment Analysis es que están muy ligados al contexto. Por ejemplo, un clasificador con gran precisión en una colección de ocio, no obtendría muy buenos resultados en una colección de política. Por ello, si nuestra colección es de una temática específica, el empleo de un clasificador ya entrenado en otra colección, obtendría resultados peores si los comparáramos con los resultados que podríamos obtener, en caso de que entrenáramos desde cero un clasificador en nuestra colección.

Igualmente, es importante mencionar que en documentos muy largos, por ejemplo, una noticia de un periódico, es complicado acertar en su polaridad. Es por ello que la mayor parte de proyectos de Sentiment Analysis utilizan como datasets mensajes cortos, como pueden ser reviews de Amazon o tweets. Por este motivo, en este proyecto los documentos a tratar serán tweets.

2.1. Preprocesamiento

El preprocesamiento de los tweets es una tarea en la que no podemos analizar un estado del arte en concreto. Dependiendo de la naturaleza de la colección, el tamaño de la misma, y como veremos, del propio idioma, habrá que explorar y se realizarán unas transformaciones distintas. Estas transformaciones tendrán como objetivo normalizar y realizar una limpieza de aquellos datos considerados irrelevantes.

Un aspecto a tener en cuenta es que no todas las transformaciones se realizan con una intención pragmática; a continuación se procederá a explicarlo con un ejemplo. La eliminación o normalización de los nombres de usuarios se realiza en casi todos los proyectos. Desde un punto de vista ético, resulta lógico que un clasificador de este estilo no debería tener sesgos hacia usuarios, pero, desde un punto de vista pragmático, en ciertas colecciones, se podría obtener un mayor acierto dejándolos intactos. Habrá colecciones donde algunos usuarios serán mencionados reiteradas veces en tweets positivos y, por ello, se convertirían en un factor determinante para clasificar un tweet como positivo.

Un preprocesado que siempre se lleva a cabo es la tokenización, que consiste en separar correctamente las palabras de un documento e introducirlas como elementos en una lista. Este proceso es necesario para poder utilizar los documentos en la mayoría de algoritmos de clasificación.

En relación con la tokenización, las transformaciones, es decir, los preprocesamientos más frecuentes, se pueden dividir en los dos grupos siguientes:

- eliminación de tokens: un token es un conjunto de caracteres que actúan como una unidad y, que se pueden encontrar entre dos espacios o, entre un espacio y símbolos de puntuación. Se eliminan tokens, ya que no se les considera de relevancia, no se quieren tener en cuenta, se consideran ruido para la colección o porque hay demasiados. De este tipo de transformaciones encontramos las siguientes:
  - Eliminación de tags de usuarios.
  - Eliminación de urls.
  - Eliminación de números.
  - Eliminación de caracteres no alfanuméricos: en ciertas ocasiones los caracteres “¿‘, “¡‘ se mantienen ya que suelen tener una polaridad fuerte.
  - Eliminación de palabras poco frecuentes.
  - Eliminación de stopwords: las stopwords son aquellas palabras que se consideran irrelevantes, ya que aparecen con mucha frecuencia en cualquier ambiente. En sentiment analysis es crucial no contabilizar como stopword la palabra “no“. También se ha visto como relevante no eliminar otros stopwords tradicionales como pronombres personales, que suelen indicar cercanía y aparecen en contenidos positivos o, también, por ejemplo, adverbios de cantidad como “poco“ o “mucho“.
- normalización de tokens: la mayor parte de los cambios aplicados sobre los documentos suelen pertenecer a este tipo. El objetivo es intentar que dos tokens, que en esencia significan lo mismo, se unifiquen para conseguir que el sistema los trate de la misma forma. Esta unificación puede suponer ciertas ventajas. Por ejemplo, imaginemos un caso donde en una colección el token ’bien’ tuviera una gran polaridad positiva y apareciera con mucha frecuencia. Si el token apareciera en un documento, obviando el resto de tokens del documento, tendríamos un

fuerte indicador para creer que es positivo. De la misma forma, en el caso de que el token 'BIEN', apareciera de forma muy poco frecuente, apenas incidiría en la polaridad del documento. Es por ello que ambos tokens se unificarían porque su significado es prácticamente el mismo. De este tipo de transformaciones encontramos las siguientes:

- Conversión a minúscula de todos los tokens.
- Lematización: el proceso de lematizar consiste en convertir los verbos a infinitivo y, los sustantivos y adjetivos, a la forma masculino singular.
- Stemming: el proceso que consiste en transformar una palabra en su raíz léxica.
- Transformación al mismo token de todos los emojis de tipo ASCII positivos, negativos o neutros. Ejemplos de este tipo de emoji serían: "xD", ":)" , "D:".
- Substitución de abreviaciones por las palabras originales.
- Eliminación de repeticiones innecesarias, por ejemplo, el token "bieeeen" se convertiría a "bien".

Dependiendo del tamaño de la colección se suele tener en cuenta lo siguiente:

- Para colecciones pequeñas (miles de tweets): se intenta eliminar la menor información posible, debido a la limitación de recursos.
- Para colecciones medianas (decenas de miles de tweets): en este caso intermedio habrá que probar y elegir entre realizar un preprocesado más agresivo o uno que elimine menor cantidad de información.
- Para colecciones grandes (centenares de miles de tweets): se suele realizar un procesado bastante agresivo, ya que con una colección tan grande el tiempo de entrenamiento para los clasificadores puede ser muy limitante. De igual forma, al tener un mayor número de documentos, los clasificadores pueden aprender con una mayor eficacia la polaridad de los tokens, tanto de forma independiente, como en relación con otros tokens. Esto produce excelentes resultados.

El idioma es un factor clave a la hora de realizar un preprocesado correcto e influye en todas las transformaciones explicadas con anterioridad. En español, aunque los recursos disponibles son bastante más reducidos que en inglés, todas las transformaciones anteriores se pueden realizar de forma correcta, salvo la de stemming. Dicha transformación es bastante complicada y, la gran complejidad gramatical del castellano, dificulta que se realice correctamente.

2.2. Modelado del lenguaje

Una vez que se tienen los documentos tokenizados, hay que encontrar la forma de pasar las listas de tokens a los distintos algoritmos de clasificación. Existen fundamentalmente dos técnicas, ambas igual de importantes: las basadas en la utilización de *sparse vectors* o *dense vectors*.

Los sparse vectors son vectores que se presentan, en su mayor parte, con múltiples valores a 0, ya que cada posición representa una palabra de la colección y, cuando tienen un valor numérico, éste representa la frecuencia de esa palabra en el documento. La longitud total de los sparse vectors será igual a la cantidad de tokens distintos que haya en toda la colección. La posición que un token toma dentro del vector no es de importancia pero normalmente esta determinada por una ordenación

alfabética.

Los dense vectors (también denominados word embeddings) son vectores con una dimensión fija en la que cada posición toma un valor. Siguen una filosofía distinta: La idea es que cada palabra de la colección puede ser representada por un vector en el espacio. De esta forma, palabras similares se agrupan en regiones similares del mismo.

Tradicionalmente, siempre se han utilizado sparse vectors para representar los documentos, pudiendo ser útil emplear una ponderación mediante la medida Term Frequency - Inverse Document Frecuency (tf-idf). Esta ponderación dará un peso menor a las palabras más frecuentes y un peso mayor a las de menor frecuencia. Las librerías de Machine Learning actuales se encuentran optimizadas para operar con este tipo de matrices poco densas. Por este motivo, siguen siendo utilizados y son, para los algoritmos de Machine Learning tradicionales, la opción a utilizar, si bien algunos de estos algoritmos también consiguen muy buenos resultados con dense vectors.

Los dense vectors están muy ligados al estado del arte actual de sentiment analysis y de NLP. Su uso está ligado a deep learning. Hay dos formas de construir estos dense vectors. La primera, utilizando algoritmos no supervisados como *Word2vec* o *FastText* (en su versión no supervisada), y la segunda, permitiendo que el modelo de red neuronal utilice una capa, llamada normalmente de *embedding*, para que aprenda por si mismo el vector que representa a cada uno de los tokens.

Word2vec es un algoritmo que permite aprender la representación vectorial de una palabra dentro de una colección. Se consigue guardando los pesos de una red neuronal entrenada bajo los algoritmos explicados en [1]. FastText es muy similar a Word2Vec. La diferencia radica en que Word2Vec trata al token del que se quiere aprender su representación vectorial de forma individualizada y, que el segundo, utiliza n-gramas formados por los caracteres del token. De esta forma, puede aprender características internas de las propias palabras (para más información acudir a [2]). Por último, la capa de Embedding es una capa de entrada a las redes neuronales que permite que, durante la fase de entrenamiento de la red, además de actualizar sus pesos, se actualice también el vector de cada uno de los tokens del documento entrante. Para ello esta capa almacena una matriz donde guarda las representaciones vectoriales de cada token de la colección, que serán seleccionados y modificados cuando un documento que les contenga aparezca como input a la red.

2.3. Modelos de clasificación

El estado del arte de las estrategias de clasificación depende como hemos visto, en gran medida, de la propia colección. No obstante, se ha observado de forma reiterada en este tema que ciertos modelos son mejores que otros. Principalmente encontramos las cuatro siguientes estrategias de clasificación:

- Diccionarios de polaridad: este tipo de diccionarios contienen las palabras más comunes de un idioma y una etiqueta de polaridad que indica si esa palabra es positiva, negativa o neutra. Empleando este tipo de diccionarios y

guardando para cada documento su cantidad de palabras positivas, negativas y desconocidas, mediante modelos de Machine Learning, se puede encontrar la polaridad del mismo. Este tipo de diccionarios son muy abundantes en inglés, sin embargo, apenas hay diccionarios adaptados para el español y la calidad de los mismos es muy baja. Dicho método de clasificación resulta útil en colecciones pequeñas.

- Clasificadores de Machine Learning tradicionales: entre ellos se encontrarían los algoritmos de aprendizaje automático sin incluir las redes profundas. Dentro de los mismos destacan los siguientes:
    - Naive Bayes: es un algoritmo muy sencillo y muy rápido de entrenar fundamentado en el Teorema de Bayes. Ha quedado demostrado que para este tipo de proyectos obtiene buenos resultados. De sus versiones, las que más destacan son Multinomial Naive Bayes y Bernoulli Naive Bayes.
    - Regresión logística: es un algoritmo que, aunque tarda más en ser entrenado, resulta de los más potentes en clasificación de texto. Se puede encontrar de forma sistemática en muchos proyectos y, generalmente, funciona bien para todas las colecciones. Permite encontrar relaciones entre las distintas palabras.
    - Máquinas de vectores de soporte (en inglés SVM): es un algoritmo, que al igual que el anterior, necesita mucho tiempo en ser entrenado, pero que también suele ofrecer buenos resultados.
- Modelos de redes profundas (deep learning): actualmente para una colección de medio-gran tamaño son los métodos más empleados. Están ocupando la mayoría de las propuestas en muchos ámbitos y, en el procesamiento del lenguaje natural, no ha sido distinto. Dentro de los modelos propuestos para estas redes, destacan las siguientes:
    - Convolutional Neural Network (CNN): su sistema de convoluciones permite, en cierta forma, ir dividiendo el documento y examinándolo por partes, como se hace con n-gramas.
    - Long Short-Term Memory (LTSM) y Gated Recurrent Unit (GRU): las redes recurrentes de este tipo pueden llegar a aprender como se relacionan las palabras de una colección porque “recuerdan“ el contexto de la misma.
    - Modelos que se basan en combinaciones y variaciones de las anteriores.
- FastText: este modelo ya se ha mencionado anteriormente como generador de dense vectors de forma no supervisada. No obstante, el modelo también se puede emplear como un clasificador con entrenamiento supervisado. Obtiene resultados iguales o algo inferiores a los modelos de deep learning, pero el tiempo de entrenamiento es de varias magnitudes menor. Para más información acudir al artículo [3].

3

Diseño y Desarrollo

Todo el proyecto se ha desarrollado empleando Python3. Este lenguaje de programación de alto nivel resulta muy útil para programas de propósito general y es altamente empleado en proyectos de Machine Learning. A continuación, en este apartado, se van a describir de manera resumida la forma en la que se desarrollaron los distintos módulos y programas para la realización de cada una de las partes del proyecto: la descarga de los datasets, la aplicación de preprocesado y el proceso de clasificación.

3.1. Descarga de los datasets

Conviene señalar que para poder desarrollar este proyecto no se tuvo mucha elección en cuanto a qué datasets emplear. Públicamente en la Web se encuentran sólo las dos siguientes colecciones de carácter general de un tamaño medio:

- **Dataset de carácter general del evento TASS2012 de la Sociedad Española para el Procesamiento del Lenguaje Natural (SEPLN) [4]:** La colección tiene un tamaño de aproximadamente 70.000 tweets de usuarios públicos de los años 2011-2012. La fecha de estos tweets es relevante porque, en su momento, el tamaño máximo era de 140 caracteres y, actualmente, es del doble. En su mayoría, los tweets tienen un sesgo hacia usuarios españoles, pero también cuenta con algunos de otros países hispanoparlantes. Los distintos temas que aparecen en la colección y, con los que los usuarios se relacionan entre sí, son los siguientes: política, fútbol, literatura y entretenimiento. Los tweets pertenecen a unos 200 usuarios públicos aproximadamente. El dataset es de una gran calidad, como se podría esperar de esta sociedad que lleva años realizando eventos y talleres para avanzar en el procesamiento del lenguaje natural en español. A lo largo de esta memoria se referenciará como “TASS2012“.
- **Dataset extraído de un proyecto que incluye otros 15 lenguajes europeos:** Desgraciadamente este dataset es de muy baja calidad. Como se comenta en el artículo [5], las dos puntuaciones fundamentales que miden la calidad para el dataset en español son muy bajas. Entre estas puntuaciones se encuentra el “self-agreement“, la capacidad que tiene un anotador para poner la misma etiqueta a un tweet igual a otro que ya anotó y, el “inter-annotator agreement“, la cohesión que tienen las etiquetas proporcionadas por los etiquetadores para la colección. A lo largo de esta memoria se referenciará como “Multilingual“.

3.1.1. Descarga del dataset Multilingual

El dataset Multilingual es un documento, de tipo CSV, con tres columnas: el identificador que el tweet tiene en Twitter, la etiqueta de polaridad y el identificador del etiquetador. Con el objetivo de adquirir el texto de cada uno de los tweets, fue necesario emplear la API de Twitter. Con esta API, mediante el identificador del tweet, se permite descargar todo el contenido relacionado con el mismo incluyendo su texto. Se empleó la librería Tweetpy, que oculta al usuario el proceso de realizar llamadas REST a la API de Twitter y, por tanto, facilita su uso. Se tuvieron que resolver problemas como la imposibilidad de descargar tweets que habían sido borrados y de aquellos que pertenecen actualmente a cuentas no públicas. Una vez descargados los textos, se realizaron las transformaciones adecuadas para construir un dataset con el formato adecuado, con dos columnas, texto y etiqueta de polaridad. Se unificaron tweets que estaban repetidos debido a que el dataset contaba con varios anotadores. El criterio para unificarlos se encuentra en el algoritmo 3.1.

```
1 if numPosTags − numNegTagsN >= numNeuTags then return posTag;
2 else if numNegTagsN − numPosTags >= numNeuTags then return negTag;
3 else return neuTag;
```

Algoritmo 3.1: Algoritmo empleado para unificar tweets repetidos con varias etiquetas. Este algoritmo es más lógico que el seleccionar como etiqueta aquella que aparezca de forma mayoritaria. Por ejemplo, si un tweet tuviera seis anotaciones positivas y una neutra, no habría duda de que el tweet debería tener etiqueta positiva. Sin embargo, si el tweet tuviera tres etiquetas positivas, dos neutras y dos negativas, todo parecería apuntar a que debería tener etiqueta neutra. Es por ello que se seleccionó este método y no uno basado en mayorías.

Una vez transformado el dataset, y tras haber estado trabajando con el mismo, resultaba evidente que parte del etiquetado no se había realizado siguiendo un criterio consciente, ya que incluso tweets con aparente polaridad clara, se encontraban mal etiquetados. Además, el número de tweets de cada clase se encontraba completamente desbalanceado.

3.1.2. Descarga del dataset TASS2012

La colección es un documento XML con varios apartados. De todos los que contiene se generó un nuevo dataset, incluyendo solamente el texto y la etiqueta de polaridad. Es importante destacar que el dataset cuenta originalmente con cuatro etiquetas: “P“, “N“, “NEU“, “NONE“. La etiqueta “NONE“, según expresan sus autores, indica la ausencia de sentimiento. Sin embargo, tras observar manualmente tweets con etiquetas “NEU“ y “NONE“, apenas se encontraron diferencias. Además, la cantidad de tweets con etiqueta “NEU“ era muy pequeña, aproximadamente 2.000 tweets. Esta etiqueta en otros datasets disponibles en la web, como por ejemplo la del dataset Multilingual, no suele aparecer, ya que su diferencia con la “NEU“ es difícilmente distinguible y no suele resultar efectivo realizar distinciones

tan puntillosas. Es por ello, que se unificaron ambas etiquetas bajo la de “NEU“. En este dataset, no hubo necesidad de llevar a cabo ninguna limpieza como en el anterior, ya que a diferencia del Multilingual, los tweets se encontraban etiquetados una única vez.

Este dataset será el empleado en este proyecto y con el que se comprobará la eficacia de los distintos tipos de preprocesados aplicados y de los clasificadores. Por tanto, a partir de este punto, hasta el final de la memoria, cuando se mencionen las palabras dataset o colección, se estará haciendo referencia al mismo.

3.2. Preprocesamiento del texto

Para realizar el preprocesado del texto se creó un módulo, llamado Preprocessor, que ejecutaba de forma secuencial las siguientes transformaciones:

1.– Transformación de las urls al token “<url>“.
2.– Transformación de los nicks al token “<nick>“.
3.– Transformación de aquellos emojis ASCII detectados a los tokens “<emoji_p>“, “<emoji_n>“, “<emoji_neu>“.
4.– Transformación de las risas detectadas a los tokens “<jaja>“, “<jeje>“, “<jiji>“ y “<jojo>“.
5.– Tokenización del texto.
6.– Conversión de los caracteres alfabéticos a minúsculas.
7.– Borrado de caracteres repetidos de forma innecesaria.
8.– Corrección de abreviaciones de longitud uno y dos.
9.– Eliminación de caracteres no alfanuméricos exceptuando emojis (Unicode) y los símbolos de exclamación e interrogación.
10.– Borrado de números y de tokens que contienen números.
11.– Lematización del texto: este es un cambio con gran impacto en la colección.
12.– Eliminación de stopwords: este cambio también tiene un gran impacto en la colección.

El módulo permite elegir si aplicar un preprocesado suave (transformaciones 1-10) u otro más fuerte al dataset (todas las transformaciones). La diferencia entre ambos tipos de preprocesados se puede observar en el cuadro 3.1.

Asimismo, el módulo contiene una transformación que permite concatenar el token “no“ al siguiente token que le preceda mediante “_“, inspirado en la idea propuesta en [6]. Dicha transformación, aunque no sea del todo intuitiva, puede llegar a ser beneficiosa. Lo que pueden aprender los clasificadores por cómo se efectúa el entrenamiento es la polaridad de las palabras y, en función de estas polaridades, clasificar un documento como una clase u otra. La palabra “no“, que aparece más usualmente en la clase negativa, también aparece en el resto. Por ejemplo, en el dataset “no“ es un token con una polaridad negativa suave. Al concatenarlo, se crean nuevos tokens de los que el clasificador puede aprender su polaridad. Esto implica que el conjunto de tokens “no quiero“ tiene un polaridad negativa

bastante inferior a la del token “no_quiero“. De ahí que se trate de una transformación útil.

1. Muy buenas noches followercetes, mañana va a ser un día bastante mitico para mi, ya os contare... http://t.co/U4obbEqe

- *procesado fuerte:* ***bueno noche followercetes mañana ir ser día bastante mítico contar <url>***
- *procesado suave:* ***muy buenas noches followercetes mañana va a ser un día bastante mitico para mi ya os contare <url>***

2. La felicidad no esta en los grandes anhelos , sino con pequeñas cosas que ocurren todos los días. Buenas noches , mañana mas ;-))

- *procesado fuerte:* ***felicidad no grande anhelo sino pequeño cosa ocurrir día bueno noche mañana mas <emoji_p>***
- *procesado suave:* ***la felicidad no esta en los grandes anhelos sino con pequeñas cosas que ocurren todos los días buenas noches mañana mas <emoji_p>***

Cuadro 3.1: En este ejemplo se pueden ver dos tweets del dataset transformados mediante el preprocesado suave y el fuerte.

A continuación, en los siguientes apartados se detalla cómo se desarrollaron estas transformaciones y que nuevos módulos se crearon. Para más información acerca de la relación entre los módulos acudir al apéndice A.1.

3.2.1. Normalización de urls y nicks

Normalmente, lo que se suele realizar, es eliminar tanto urls como nicks. Sin embargo, se decidió normalizarlos porque su aparición en las distintas clases no es del todo equitativa como se verá en el apartado de resultados. Tanto el nick como la url siguen expresiones regulares fijas en Twitter y es fácil normalizarlos.

3.2.2. Normalización emojis ASCII

Se decidió emplear esta normalización porque en el análisis de tokens de longitud dos se vió que el emoji “:)“ era muy frecuente y, por tanto, se llegó a la conclusión de que podría haber bastantes más con un formato parecido. Conviene normalizar los emojis ASCII porque suelen tener una polaridad muy marcada. Esta tarea se incluyó en el módulo denominado EmojiNormalizer.

El primer paso fue agrupar los emojis en tres grupos: positivos, negativos y neutros. No es la forma más óptima, puesto que al efectuarlo manualmente, se pueden llegar a realizar unificaciones que posiblemente no sean del todo correctas y terminan presentando un componente subjetivo. Sin embargo, como se verá en el apartado cinco, esta normalización habría resultado eficaz.

Se pueden normalizar porque en general, en cuanto a formato, hay dos tipos de emojis. Un primer tipo emplea un único símbolo para los ojos y otro para la boca y, un segundo, emplea dos símbolos para los ojos y, en medio, otro símbolo para la boca. Ambos pueden añadir un símbolo para la nariz. Asimismo, se les añaden repeticiones, como por ejemplo ":DDDDD", o recursividad, como ":):):):):)", para hacerlos más enfáticos. Los recursivos se identificaron como un único emoji para que no tuvieran un peso excesivo en los tweets donde aparecieran.

Para identificarlos se emplearon expresiones regulares. El formato de las expresiones regulares desarrolladas para cada grupo es similar, pero presentan ciertas diferencias destacables. Una diferencia es que los emojis positivos se pueden separar, incluso si van unidos a una palabra. Por ejemplo, "Buen díaaa:)))" pasaría a ser "Buen día <emoji_p>". Sin embargo, con los negativos no se pudo llevar a cabo de forma segura esta tarea porque se convertían en expresiones como 'PD:" a "P <emoji_n>". Por tanto, se decidió que las expresiones regulares para los negativos y neutros no fueran tan flexibles.

Se podría discutir si igual hubiese sido preferible dejar los emojis intactos para que los clasificadores aprendiesen el significado de estos tokens por si solos. Sin embargo, esto daría lugar a dos problemas principales que surgirían a raíz de aspectos ya mencionados:

- Los emojis presentan repeticiones y recursividad. Por tanto, aunque semánticamente ":)" y ":))))" sean casi lo mismo, al segundo emoji apenas se le daría importancia al ser muchísimo más infrecuente.
- Los tokenizadores deben separar los caracteres no alfanuméricos de las palabras. Por ejemplo, "además," pasaría a ser "[además , ',']". Algunos tokenizadores sí detectan emojis sencillos como ":)", pero no suelen hacerlo con las otras variantes mencionadas. Esto causaría que se separasen los emojis en varios tokens, perdiendo completamente su valor semántico.

3.2.3. Normalización de risas

Esta transformación, similar a la anterior, se resolvió creando un módulo, denominado LaughNormalizer, que emplea también expresiones regulares. Se decidió crear cuatro grupos de risas porque cada una tiene una connotación algo distinta. El análisis sobre su polaridad se verá más adelante. Las risas pueden aparecer de dos formas: como una unidad conjunta o separada por espacios. Además, al aparecer dichas risas en mensajes de naturaleza espontánea y poco meditada, suelen presentarse con muchas erratas. Las erratas más comunes son el cambio de la 'j' por la 'h' y la 'a' por la 's', lo que se debe a que son las letras que se encuentran más juntas en el teclado y, aunque ciertas personas detecten este fallo ortográfico, no suelen corregirlo, ya que las faltas dotan al mensaje de una esencia más espontánea y distendida. Al igual que para los emojis, toda la comprobación para ir construyendo las expresiones regulares, se tuvo que llevar a cabo manualmente.

3.2.4. Creación de un wrapper en Python para Stanford CoreNLP

El departamento de procesamiento del lenguaje natural de Stanford cuenta con un servidor Java [7] de acceso público, que realiza ciertas operaciones del lenguaje natural para varios idiomas, incluido el español. Dicho servidor tiene para el español el proceso de análisis sintáctico y tokenización. Para poder utilizar el servidor de Stanford, se desarrolló un módulo, denominado WrapperCoreNLP, que actuó como wrapper del servidor. Dicho módulo usa un paquete Python, desarrollado por Stanford, que permite tratar al servidor Java como si fuera una clase. El módulo propio desarrollado contiene dos métodos que realizan y parsean la respuesta JSON del servidor, uno para la tokenización y, el otro, para el análisis sintáctico.

3.2.5. Tokenización

La tokenización, como se ha explicado antes, es el proceso que consiste en transformar una frase a una lista de tokens. No consiste en separar las palabras por espacios en blanco, sino en dividir el texto en unidades (tokens) con sentido. Se va a ilustrar la diferencia con un ejemplo. La frase “Al fin, se acabó.´´ separada por espacios en blanco se convierte en la lista [“Al“, “fin,“ ,“se“ ,“acabó.“]. Tokenizando se convierte en la lista [“Al“ ,“fin“ ,“,“ ,“se“, “acabó“, “.“]. Para llevar a cabo la tokenización se emplea el modulo WrapperCoreNLP.

3.2.6. Borrado de caracteres innecesarios

El hecho de poner repeticiones de caracteres de forma innecesaria es algo que se realiza con bastante frecuencia en las redes sociales. Por ejemplo, “que guaaaaay“. Sin embargo, eliminarlo no es trivial por los siguientes motivos:

- Las consonantes r, l, c, n aparecen de forma natural repetidas dos veces en muchas palabras.
- Las vocales aparecen repetidas de forma frecuente en derivaciones de verbos y en algunos adjetivos.

Para eliminarlas correctamente lo que se hace es emplear los ficheros realizados para el módulo Lemmatizer (apartado 3.2.8), con el fin de comprobar si la palabra con letras repetidas, aparece como una palabra derivada o como un lema, en cuyo caso no se eliminaría la repetición.

3.2.7. Eliminación de abreviaciones

Para tal fin, se examinó la colección para ver qué tokens, de longitud uno y dos, aparecían con mayor frecuencia. Se descubrió que muchos de ellos eran abreviaciones de otras palabras, como por ejemplo “k“,“d“ y “xq“. Una vez detectadas las abreviaciones más comunes se revirtieron a su palabra

original. Esta transformación es necesaria para realizar correctamente lematización y eliminación de stopwords.

3.2.8. Proceso de lematización

El proceso de lematización es un proceso que se puede resolver principalmente de dos formas, mediante aprendizaje supervisado o usando diccionarios de lemas y derivados con un analizador sintáctico. Se ha llevado a cabo de la segunda forma. Para ello es fundamental el uso de un analizador sintáctico, ya que si no conociéramos el grupo sintáctico al que pertenece una palabra, no podríamos saber cuando la queramos lematizar, a qué convertirla. Por ejemplo, la palabra “cocinas“ hay que convertirla a “cocinar“ o “cocina“, dependiendo de cómo aparezca sintácticamente. Para la realización de esta tarea se crea un módulo denominado Lemmatizer.

El diccionario de lemas y derivados empleado, que se descargó del repositorio [8], tiene dos columnas. La primera, representando los lemas y, la segunda, las palabras derivadas. Se tuvo que realizar un ajuste especial a este fichero porque se encontró que, en ciertos casos, existían recurrencias, de forma que el lema de una palabra derivada era a su vez una palabra derivada de otro lema. También, como se habían observado bastantes fallos ortográficos debidos a la ausencia de tildes, se añadieron estas palabras acentuadas sin acentuar, para poderlas convertir a su lema. Posteriormente, se dividió el fichero en verbos y no verbos.

Al inicializar el módulo Lemmatizer dichos ficheros se mapean a atributos, de tipo diccionario, dentro del mismo. Con este nuevo módulo y con WrapperCoreNLP, para etiquetar sintácticamente la frase, ya se podía llevar a cabo el proceso de lematización. El primer paso es etiquetar sintácticamente las palabras. Dependiendo de esta etiqueta se busca el lema en el diccionario de los verbos o de los no verbos. Se va a matizar con un ejemplo. Para lematizar la palabra “trabajaba“ el módulo WrapperCoreNLP asigna al token la etiqueta de verbo. Con esta etiqueta, el módulo Lemmatizer, busca si la palabra está contenida en su diccionario de verbos. Si aparece en dicho diccionario, significa que era una palabra derivada y se sustituye por su lema, en caso contrario, se mantiene intacta. Finalmente, repitiendo este proceso con todas las palabras, se devuelve la frase lematizada.

Si se aplican transformaciones, como las explicadas anteriormente, hay que realizar un preprocesado auxiliar, distinto al ya mencionado, a la frase antes de pasarla al lematizador para no reducir la eficacia del mismo. Se tuvieron en cuenta los siguientes cambios: el token <url> se cambia por la palabra url, el token <nick> se cambia por un nombre propio y, los tokens de los emojis y risas, son eliminados. Una vez lematizada la frase, se revierten estos cambios de preprocesado. Es importante destacar que la lematización puede fallar por dos motivos: El primer motivo es que el analizador sintáctico falle a la hora de poner las etiquetas. En este caso puede que no se encuentre el lema de la palabra o que se le asigne un lema incorrecto. El segundo caso es que el lematizador tenga que procesar una palabra que pueda tener varios lemas. Esto es lo que ocurre con la palabra “fuí“, que puede pertenecer

al verbo “ser“ y al verbo “ir“. Sin información contextual es imposible hacer la correcta lematización de dicha palabra. Cuando se dan estos casos, se cogerá de forma arbitraria uno de los dos lemas. Se comprobó que no se producen problemas para el 99.5 % de todas las palabras del diccionario.

3.2.9. Eliminación de stopwords

Para eliminar stopwords se empleó la librería NLTK [9] que contiene una lista de stopwords para el español. Estas palabras consisten mayormente en adverbios, pronombres, y preposiciones. De la lista de stopwords que ofrece la librería, se han eliminado las palabras “no“, “ni“, “poco“, “mucho“ y “nada“ que no se consideran como stopwords. Cabe destacar que la librería empleada tenía dos erratas en los pronombres “vosotros“ y “vosotras“. Dicho error ha sido notificado a los creadores de la librería.

3.2.10. Datasets preprocesados creados

Empleando el módulo de preprocesamiento desarrollado, se crearon seis datasets preprocesados distintos a partir del dataset TASS2012 original, que se denominaron de la siguiente forma:

- **splitted:** tokens separados simplemente por espacios en blanco.
- **tokenized:** tokenizando correctamente como se ha explicado en el apartado 3.2.5.
- **soft-processed:** aplicando sólo las técnicas consideradas suaves (en la subsección 3.2 del 1 al 10).
- **strong-processed:** aplicando todas las técnicas de preprocesado.
- **no-appended-soft-processed:** aplicando sólo las técnicas consideradas como suaves y concatenando el “no“ delante de la palabra que le precede.
- **no-appended-strong-processed:** aplicando todas las técnicas de preprocesado y concatenando el “no“ delante de la palabra que le precede.

Se decidió no generar más tipos de datasets preprocesados porque con estos seis se representaban las opciones más relevantes del preprocesado.

3.3. Clasificación del texto

En este apartado se va a aportar información acerca de como se desarrollaron las fases del proceso de clasificación: la división en partes del dataset, la vectorización de los documentos, la investigación de los mejores modelos de clasificación, tanto los de Machine Learning tradicionales como los de deep learning, y el análisis del funcionamiento interno de dos de los mejores modelos probados. Para esta parte resultaron imprescindibles las bibliotecas de Scikit, Keras y Gensim. Scikit es una librería muy famosa de Machine Learning que contiene una cantidad variada de modelos de clasificación y otras herramientas de Data Science. Keras es hoy en día una de las mejores librerías para la construcción

de forma simple de modelos de deep learning. Por último, Gensim es una librería que implementa los mecanismos más populares de creación de dense vectors, como Word2Vec y Doc2Vec. Resultaron muy útiles los Jupyter Notebooks que se emplearon con frecuencia para realizar pruebas llevadas a cabo en esta parte del proyecto.

3.3.1. Preparación inicial de la clasificación

Para cada uno de los datasets preprocesados se realizó de forma aleatoria su división en un conjunto de train/test (70 %) y otro de validación (30 %). El conjunto train/test se emplea para entrenar, configurar y comprobar la eficacia de los distintos tipos de clasificadores. El conjunto de validación se emplea con el objetivo de conseguir unos resultados finales de la eficacia de los modelos con mejor rendimiento en el conjunto de train/test. La división de las colecciones se ejecuta al azar, pero siempre se utiliza la misma semilla. De esta forma, la división aleatoria es igual para todas las colecciones preprocesadas. Si no se lleva acabo de este modo, las pruebas que se realizan más adelante no serían concluyentes.

3.3.2. Vectorización con sparse vectors

Los tweets tokenizados tienen longitud variable y los clasificadores deben recibir vectores de tamaño fijo. Una forma de conseguir estos vectores es vectorizar los tweets con el enfoque de sparse vectors. Para ello, se usaron las clases de Scikit CountVectorizer y TfidfVectorizer [10], similar al primero, pero aplicando la ponderación tf-idf. Estos vectorizadores, construyen su estructura de sparse vectors a raíz de los datos de train y, posteriormente, los datos de test se transforman empleando el vectorizador ya entrenado. Aquellas palabras de test que no aparezcan en el vectorizador ya entrenado, serán ignoradas. Se puede ver un ejemplo del funcionamiento en el cuadro 3.2. Al final del proceso se obtiene un sparse matrix con número de filas, igual a la totalidad de los documentos y, columnas, igual a la cantidad de tokens distintos. Por ejemplo, para el preprocesado tokenized la matriz tendría apróximadamente una dimesión de 70.000x100.000. Otra clase de Scikit usada fue Pipeline [11]. Es un objeto que almacena un vectorizador y un clasificador. Es relevante ya que nos permite no tener que hacer train/test del vectorizador y del modelo de clasificación de forma separada. En resumen, envía la salida del vectorizador directamente al modelo.

3.3.3. Vectorización con dense vectors

El enfoque alternativo de vectorización es construir dense vectors. Para crearlos se utilizó la librería Gensim. En este proyecto se han explorado los algoritmos Word2Vec y Doc2Vec, usando las clases de Gensim, que reciben el mismo nombre que los algoritmos. Para más información revisar [12] y [13].

Colección:

1.– "Hola", "buenos", "buenos", "días"

2.– "Tienes", "unos", "buenos", "amigos"

3.– "Unos", "días", "buenos"

Las columnas de los sparse vectors contienen las palabras ordenadas alfabéticamente. En este caso serían: Hola, Tienes, Unos, amigos, buenos, días, unos.
Resultado de aplicar CountVectorizer:

1.– 1 0 0 0 2 1 0

2.– 0 1 0 1 1 0 1

3.– 0 0 1 0 1 1 0

Cuadro 3.2: Ejemplo del funcionamiento de CountVectorizer para una colección formada por tres frases.

Con Word2Vec cada token se transforma en un dense vector (word embedding). Por este motivo, un tweet se compone por tantos dense vectors como tokens tenga, incluyendo además tokens de relleno porque todos los tweets han de tener longitud fija. Siguiendo el ejemplo de los sparse vectors, con una dimensión de 128 como tamaño de vector por token, con 35 tokens por tweet, la matriz sería de dimensión 70.000x35x128. En contraste, Doc2Vec genera un dense vector por cada tweet y por tanto la dimensión, con tamaño de vector de 128, sería de 70.000x128. A continuación, se van a explicar brevemente los hiper-parámetros más importantes y los valores que se les han dado para configurar adecuadamente estos algoritmos:

- **Tamaño de ventana:** Este parámetro indica cuántas palabras se van a observar alrededor de la palabra a estudiar. El tamaño de palabra, como se puede ver en [14], hay que variarlo dependiendo del objetivo. Si la intención es aprender sinonimia y encontrar otras palabras que cumplan una funcionalidad parecida, se usará un tamaño de ventana pequeña. Con un tamaño de ventana más grande se agruparían las palabras en función de los tópicos. En nuestro caso, se utilizó un tamaño de ventana pequeño de dimensión dos porque nos interesa la primera opción.
- **Número de muestras negativas:** estas técnicas emplean el método conocido como negative sampling [15]. Este parámetro indica cuantas palabras de ruido se introducirán para entrenar el vector de una palabra. Se utilizaron valores entre 9 y 20.
- **Tamaño del vector que tendrá la palabra**. Para colecciones pequeñas se indica que la dimensión no debe ser muy grande. Se utilizaron 128. En proyectos con una cantidad bastante superior de documentos se suele utilizar entre 300 y 500.
- **Algoritmos de entrenamiento:** Word2vec cuenta con dos algoritmos de entrenamiento: skip-gram y CBOW. CBOW aprende mejor las palabras más frecuentes y necesita de un mayor número de palabras, mientras que skip-gram aprende mejor las palabras poco frecuentes. Doc2Vec tiene dos algoritmos análogos a los anteriores, PV-DM para CBOW y PV-DBOW para skip-gram. Se probaron ambas soluciones.

No obstante, la creación de los dense vectors no tiene porque hacerse de manera no supervisada,

las redes profundas pueden aprender por sí solas los vectores de cada palabra. Para ello usan una capa de embedding, denominada en Keras, Embedding. En el apéndice B.1 se puede ver en detalle su uso.

3.3.4. Clasificadores de Machine Learning tradicionales

Para la prueba de los distintos modelos de clasificación tradicionales se han empleado las siguientes clases de la librería Scikit: *MultinomialNB, BernoulliNB, LogisticRegression, LinearSVC, SVC, Perceptron, MLPClassifier*, y *RandomForest*. También, resultaron de gran utilidad los métodos *cross_validate* (validación cruzada) y *train_test_split* (validación simple) para realizar particiones del dataset creado para train/test. Se usaron de la librería las métricas *f1_score* y *accuracy_score*, para medir la eficacia de los modelos. Se proporcionará más información acerca de las métricas en la siguiente sección. Por último, mencionar que la clase *ParameterGrid* fue conveniente para realizar búsqueda en anchura y configurar los hiper parámetros de ciertos modelos. La información detallada sobre los modelos de clasificación, además de los otros métodos mencionados, se encuentra en la API de la librería [16].

3.3.5. Clasificadores de deep learning

Para probar modelos de deep learning se empleó la librería Keras, que permite crear modelos LSTM y GRU de forma sencilla [17]. En el entrenamiento de este tipo de redes, se emplea el conjunto de train, para que ajusten sus pesos, y el de test, para analizar la eficacia de la red en cada época. Las redes deben de ser entrenadas hasta un punto límite. Una vez superado el mismo, pasan a estar sobreaprendidas y a reducir su tasa de acierto en test. Para monitorizarlas, se añadió la técnica denominada fast stopping. Esta técnica finaliza el entrenamiento de la red cuando detecta que, tras haber transcurrido un número límite de épocas, no ha conseguido aumentar su precisión en el conjunto de test. También, se introdujeron puntos de control donde se guarda el estado de la red. De esta forma, cuando la red comience a sobreentrenar y se pare con fast stopping, no se obtiene un modelo con baja eficacia, sino el mejor modelo entrenado hasta el momento. Más información puede encontrarse en [18]. En el apéndice B.1 se muestra un ejemplo completo de la creación de una de estas redes.

3.3.6. Análisis de los mejores modelos de clasificación

Encontrados los modelos que resolvían de manera más eficaz la tarea de clasificación, se decidió analizar dos de los que habían ofrecido mejores resultados. Estos fueron la regresión logística y la red neuronal GRU. Los métodos de análisis que se emplearon se explican a continuación.

Para la regresión logística, se examinaron los valores de los pesos de cada uno de sus tres vectores de pesos, uno por cada clase. Cada vector tiene tantas columnas como tokens distintos en la colección,

ya que este modelo recibía sparse vectors como entrada. El análisis realizado consistió en extraer los coeficientes de cada vector, llevar a cabo la exponencial de cada uno, y encontrar qué palabras tenían una mayor ponderación. Se quiso probar la intuición de que, las palabras más frecuentes, si además presentan mucha cohesión con una clase en particular, deben ser las que aparecieran con una ponderación alta. Para ello se desarrolló una métrica que mide la cohesión de las palabras más frecuentes de la colección. La métrica desarrollada, que recibirá el nombre en esta memoria de WCCo (word class cohesion), sigue la ecuación 3.1. Como se puede apreciar es equivalente a la P(token aparezca en un documento de la clase). Por tanto, cuanto más valor, mayor cohesión.

$$WCCo(token, class) = \frac{frec(token, class)}{frec(token, dataset)} \tag{3.1}$$

A raíz de la métrica anterior, se construye otra, denominada CCo (class cohesion), que mide la cohesión que las palabras más frecuentes tienen con respecto a su clase y al resto. Es decir, si estamos evaluando las palabras relevantes de la clase positiva obtendremos tres valores: la cohesión de esas palabras con la clase positiva, con la negativa y con la neutra. Para medir la cohesión se calcula la WCCo de las N palabras más frecuentes de cada clase. Después se utiliza la ecuación 3.2 donde "class" es la clase de la que se extraen las palabras más frecuentes y donde "output_class" indica cual de los tres valores de cada clase, mencionados anteriormente, obtendremos. De nuevo, a mayor valor de la métrica, mayor la cohesión de los tokens de una clase con la misma.

$$CCo(class, output\_class) = \sum_{token}^{n\_most\_common\_tokens(class)} WCCo(token, output\_class) \tag{3.2}$$

Es importante señalar que dichas métricas no ponderan por frecuencia, por tanto, asumen la hipótesis de que las N palabras más frecuentes, tienen la misma influencia. Si no se llevase a cabo, debido a la ley de Zipf [19], palabras muy frecuentes con apenas cohesión aparecerían con un valor muy alto. También, se decidió en este caso no tener en cuenta los stopwords.

Para la red GRU, al ser una red recurrente, el contenido de sus pesos carece de una interpretabilidad clara. Por este motivo se decidió analizar los word embeddings que la red había aprendido, ya que se sospechaba que estos vectores se encontrarían agrupados en tres secciones del espacio vectorial, una para cada clase. Con el objetivo de analizarlos se seleccionaron 900 embeddings, con las 300 palabras más frecuentes de cada clase, sin tener en cuenta stopwords. Posteriormente, estos embeddings se agruparon empleando Agrupamiento jerárquico (Hierarchical clustering), con estrategia aglomerativa, distancia coseno y tres clusters. Se utilizó la clase AgglomerativeClustering de Scikit [20].

Tanto los resultados obtenidos para la regresión logística, como para la GRU, se pueden ver en la siguiente sección.

4

Integración, pruebas y resultados

En este apartado se van a presentar y discutir los resultados obtenidos a raíz del código desarrollado con anterioridad. El apartado se va a dividir en tres partes. En la primera, se dará información acerca del dataset, en la segunda, se mostrará información acerca de datos relevantes acerca del preprocesado realizado y, finalmente, en la tercera, se proporcionará información sobre los resultados obtenidos en la clasificación.

4.1. Análisis preliminar del dataset

Las clases del dataset original se encuentran distribuidas, de forma equitativa, de la siguiente manera: de un total de 68.016 tweets, 18.026 son negativos, 24.873 neutros y 25.117 positivos.

En cuanto a la distribución de la frecuencia, de los tokens del dataset, está formado por un total de 102430 tokens siguiendo una distribución marcada por la ley de Zipf, como es lógico en un proyecto de NLP.

Por último, calculando la CCo de las 100 palabras más comunes de cada una de las clases, métrica definida con anterioridad, se genera la matriz de confusión 4.1. Se puede observar en esta matriz el hecho de que la clase con mayor cohesión es la positiva, luego la negativa y, por último, la neutra.

4.2. Análisis del preprocesado

En este apartado se va a facilitar información sobre las transformaciones más destacables aplicadas al dataset, para determinar si fueron de relevancia. Se proporcionarán solo datos acerca de las normalizaciones más agresivas y posiblemente más cuestionables. Tampoco se incidirá excesivamente, pues la clasificación revelará cuál ha sido el mejor preprocesado.

La decisión tomada de no eliminar urls y nicks, sino normalizarlos, resultó correcta como se puede ver en la figura 4.2. En ambas etiquetas no existe una distribución equitativa, pero sí un cierto desbalanceo hacia una de las clases. Esto implica que los clasificadores podrán aprender la polaridad de

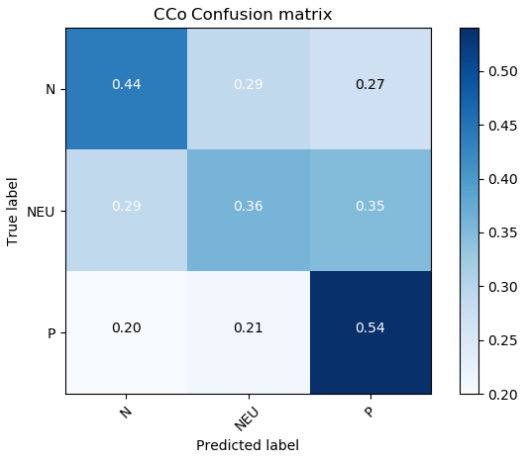


Figura 4.1: Matriz de confusión donde se puede ver la cohesión que las 100 palabras más frecuentes de cada clase tienen con respecto a la misma y con el resto.

estos tokens y utilizarlos para decidir a qué clase corresponde un documento. Por ejemplo, cuando un clasificador entrenado encuentre el token <nick> tendrá un mayor indicio para clasificar el documento que lo contiene como positivo.

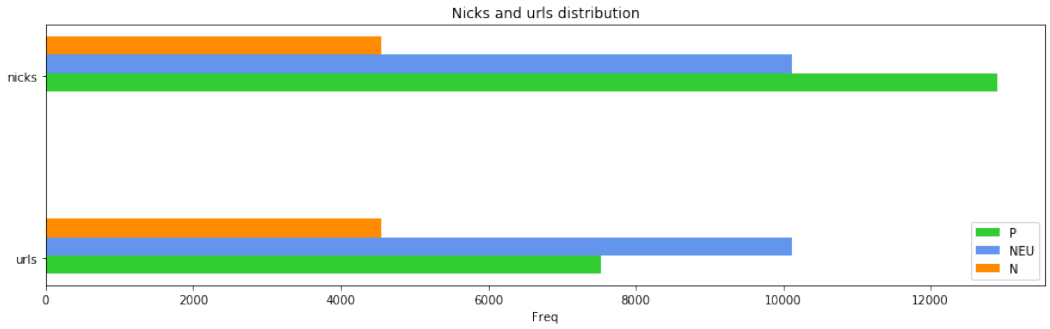


Figura 4.2: Distribución de los tokens <nick> y <url> en cada una de las clases de la colección.

La normalización de emojis ASCII resultó en la transformación de 4.000 tokens de este tipo. Los resultados de la normalización de emojis ASCII se pueden observar en la matriz de confusión 4.3. Se emplea una matriz de confusión porque se realizó una división manual de los emojis. Se concluye de estos resultados que la normalización de emojis ha sido bastante exitosa, ya que es un elemento que ayuda claramente a distinguir tweets positivos, y en menor medida, a los negativos y neutros. Se obtienen los resultados esperados. Resulta lógico que los emojis positivos aparezcan el 93 % en tweets positivos. Los emojis negativos no tienen tanta cohesión con la clase negativa (54 %). Este aspecto extraño es coherente, ya que, en cierta forma, los emojis ofrecen un carácter más distendido al contenido del mensaje y, este carácter es impropio de bastantes tweets negativos, que tienen un carácter más serio. Con los emojis neutros ocurre un comportamiento similar. Al ser los emojis de esencia distendida, surge que los neutros, que expresan sentimientos como la sorpresa, se usan más en la clase positiva.

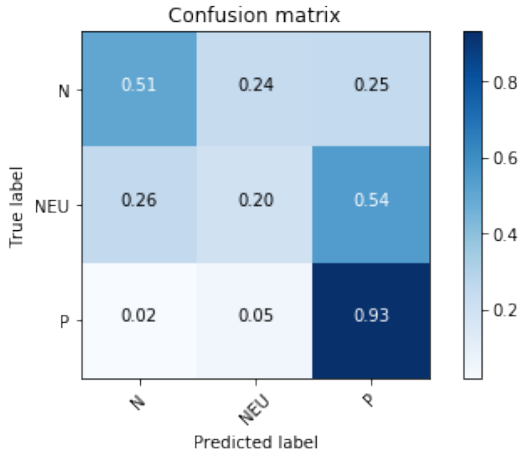


Figura 4.3: Matriz de confusión que muestra la distribución de los distintos emojis separados manualmente en cada una de las clases.

Como se puede en la figura 4.4 para la normalización de las risas se observa una naturaleza parecida a la de los emojis ASCII. La distribución de la frecuencia de las distintas etiquetas se encuentra desbalanceada, sin embargo, se decidió no unificarlas porque semánticamente no representan lo mismo. La cantidad de tokens que representan risa en el dataset es de aproximadamente 1.000. En general, muestran más cohesión en la clase positiva y, por tanto, resultan útiles para distinguirla del resto. Como podemos ver, la clase positiva cuenta siempre con un mayor número de etiquetas, seguido por la clase neutra y, muy raramente, por la negativa.

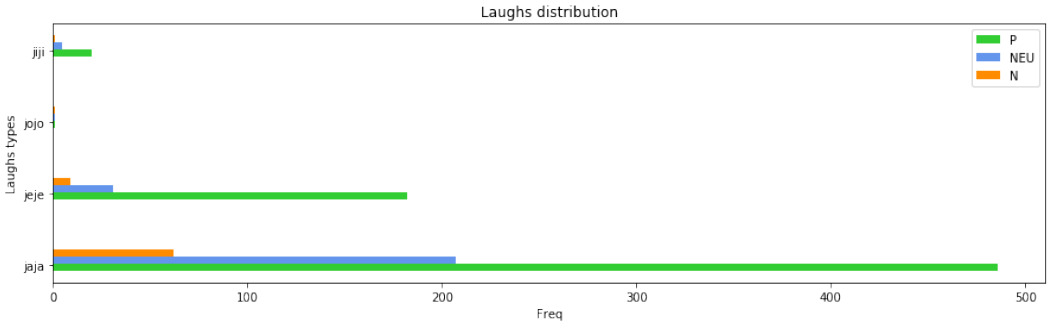


Figura 4.4: Distribución de los distintos tokens que representan risas en cada una de las clases de la colección.

Por último, concluir que el procesado suave consigue reducir la dimensionalidad del problema de 102.430 tokens a 53.172 y, el procesado fuerte, la reduce a 37.622. Esto significa que mediante la lematización se han conseguido unificar aproximadamente 15.000 palabras.

Se ha decidido no incluir un análisis individualizado de los beneficios de cada uno de los componentes del preprocesado ya que la cantidad de combinaciones distintas es abrumadora. Estas se disparan porque ciertas transformaciones deben de ser probadas a su vez con otras. Por tanto, quedan estas pruebas fuera del alcance de esta memoria a favor de incluir otro contenido de mayor relevancia.

No obstante, como se ha expresado anteriormente, se medirá la eficacia de las transformaciones de forma conjunta empleando los datasets preprocesados mencionados en la subsección 3.2.10.

4.3. Resultados de la clasificación

En este apartado se van a mostrar los resultados obtenidos de las distintas técnicas de Machine Learning. Las métricas que se emplean para comprobar la efectividad de los modelos son el accuracy, reflejado en la ecuación 4.1, y el macro average f1-score, reflejado en la ecuación 4.5. La segunda métrica es útil para conocer si los modelos tienen una buena eficacia en todas las clases porque desprecia el tamaño de las mismas al hacer la media. Para más información revisar [21]. Todos los resultados mostrados en este apartado provienen de realizar una validación cruzada con K=10 sobre el conjunto train/test, salvo que se indique lo contrario.

$$Accuracy(y,t) = \frac{1}{num\_docs} \sum_{i=1}^{num\_docs} 1(y_i == t_i) \text{ siendo y(predicciones), t(etiquetas reales)} \quad (4.1)$$

$$Precision(A,B) = \frac{|A \cap B|}{|A|} \text{ para dos conjuntos A y B} \quad (4.2)$$

$$Recall(A,B) = \frac{|A \cap B|}{|B|} \text{ para dos conjuntos A y B} \quad (4.3)$$

$$F1(A,B) = \frac{Precision(A,B) * Recall(A,B)}{Precision(A,B) + Recall(A,B)} \text{ para dos conjuntos A y B} \quad (4.4)$$

$$MacroAveragedF1(y,t) = \frac{1}{|L|} \sum_{l \in L} F1(y_l, t_l) \text{ siendo L el conjunto de etiquetas} \quad (4.5)$$

En un primer apartado, se muestran los resultados que los algoritmos de Machine Learning tradicionales, usando sparse vectors, obtuvieron para cada tipo de preprocesado. En un segundo apartado, usando el dataset con el mejor preprocesado encontrado para resolver la tarea de clasificación, se muestran los resultados para más clasificadores con sparse vectors. En un tercer apartado, se analizan los resultados obtenidos usando word embeddings y con deep learning. En un cuarto, se hace un análisis de interpretabilidad de dos de los mejores modelos. Por último, se aportan los resultados, de los tres mejores modelos encontrados en el conjunto de validación.

4.3.1. Selección del mejor tipo de procesado

Para encontrar el preprocesado más efectivo se probó cada uno de los mencionados en la subsección 3.2.10, con cuatro tipos de clasificadores de machine learning distintos, comunes en el ámbito de NLP, empleando sparse vectors. Los resultados de las pruebas se pueden ver en las siguientes figuras 4.5, 4.6, 4.7, 4.8, 4.9 y 4.10.

Model	Tf-idf	Acc	Macro avg
MultinomialNB {smoothing=yes}	no	0.6600	0.6554
BernoulliNB {smoothing=yes}	no	0.6627	0.6552
LogisticRegression {multi_class=OVR,max_iter=100}	no	0.7088	0.7078
LinearSVC {multi_class=OVR,loss=squared_hinge}	no	0.6929	0.6916
MultinomialNB {smoothing=yes}	yes	0.6553	0.6495
BernoulliNB {smoothing=yes}	yes	0.6627	0.6552
LogisticRegression {multi_class=OVR,max_iter=100}	yes	0.6918	0.6902
LinearSVC {multi_class=OVR,loss=squared_hinge}	yes	0.7009	0.6989

Figura 4.5: Resultados de varios clasificadores recibiendo el dataset “splitted“ (tokens separados por espacios en blanco). Los resultados son buenos en si mismos. En gran medida esto se debe, muy posiblemente, a la calidad del dataset.

Model	Tf-idf	Acc	Macro avg
MultinomialNB {smoothing=yes}	no	0.6598	0.6553
BernoulliNB {smoothing=yes}	no	0.6775	0.6763
LogisticRegression {multi_class=OVR,max_iter=100}	no	0.7249	0.7239
LinearSVC {multi_class=OVR,loss=squared_hinge}	no	0.6988	0.6970
MultinomialNB {smoothing=yes}	yes	0.6594	0.6545
BernoulliNB {smoothing=yes}	yes	0.6775	0.6763
LogisticRegression {multi_class=OVR,max_iter=100}	yes	0.7094	0.7081
LinearSVC {multi_class=OVR,loss=squared_hinge}	yes	0.7137	0.7119

Figura 4.6: Resultados de varios clasificadores recibiendo el dataset “tokenized“ (tokenizado correctamente). Como se puede observar el accuracy aumenta de forma general en los distintos métodos. Esto indica la importancia de realizar una correcta tokenización.

Lo primero que se puede observar de los resultados es que hay una tendencia muy similar en los seis tipos de preprocesados, siendo en todas las pruebas el peor MultinomialNB, seguido de BernoulliNB, después LinearSVC y, por último, LogisticRegression. En cuanto a la vectorización no se llegan a resultados concluyentes para determinar que una sea mejor que otra. Lo que si se observa es que LogisticRegression sin tf-idf resulta alrededor de un 1 % más efectiva. En contraste, LinearSVC aumenta su efectividad por encima del 1 % con tf-idf. Los métodos bayesianos no se ven apenas afectados por la ponderación.

El tipo de ponderación tf-idf no tiene porqué resultar siempre efectiva en este contexto. Lo que

Model	Tf-idf	Acc	Macro avg
MultinomialNB {smoothing=yes}	no	0.6684	0.6664
BernoulliNB {smoothing=yes}	no	0.6859	0.6848
LogisticRegression {multi_class=OVR,max_iter=100}	no	0.7384	0.7362
LinearSVC {multi_class=OVR,loss=squared_hinge}	no	0.7106	0.7080
MultinomialNB {smoothing=yes}	yes	0.6657	0.6628
BernoulliNB {smoothing=yes}	yes	0.6859	0.6848
LogisticRegression {multi_class=OVR,max_iter=100}	yes	0.7304	0.7293
LinearSVC {multi_class=OVR,loss=squared_hinge}	yes	0.7230	0.7212

Figura 4.7: Resultados de varios clasificadores recibiendo el dataset “strong-processed“ (preprocesado fuerte). Se consiguen mejores resultados que con la tokenización. Esto, a priori, parecería indicar que el proceso de lematización se ha realizado de forma correcta.

Model	Tf-idf	Acc	Macro avg
MultinomialNB {smoothing=yes}	no	0.6687	0.6667
BernoulliNB {smoothing=yes}	no	0.6876	0.6862
LogisticRegression {multi_class=OVR,max_iter=100}	no	0.7404	0.7383
LinearSVC {multi_class=OVR,loss=squared_hinge}	no	0.7141	0.7112
MultinomialNB {smoothing=yes}	yes	0.6671	0.6624
BernoulliNB {smoothing=yes}	yes	0.6876	0.6862
LogisticRegression {multi_class=OVR,max_iter=100}	yes	0.7292	0.7279
LinearSVC {multi_class=OVR,loss=squared_hinge}	yes	0.7282	0.7262

Figura 4.8: Resultados de varios clasificadores recibiendo el dataset “soft-processed“ (preprocesado suave). Ofrece unos resultados algo mejores al anterior.

Model	Tf-idf	Acc	Macro avg
MultinomialNB {smoothing=yes}	no	0.6688	0.6667
BernoulliNB {smoothing=yes}	no	0.6874	0.6862
LogisticRegression {multi_class=OVR,max_iter=100}	no	0.7401	0.7379
LinearSVC {multi_class=OVR,loss=squared_hinge}	no	0.7140	0.7113
MultinomialNB {smoothing=yes}	yes	0.6673	0.6642
BernoulliNB {smoothing=yes}	yes	0.6874	0.6862
LogisticRegression {multi_class=OVR,max_iter=100}	yes	0.7329	0.7318
LinearSVC {multi_class=OVR,loss=squared_hinge}	yes	0.7274	0.7255

Figura 4.9: Resultados de varios clasificadores recibiendo el dataset “no-appended-strong-processed“ (preprocesado fuerte con el “no“ concatenado). Concatenar los “no“ en el preprocesado fuerte ofrece un muy leve mejor resultado que no hacerlo.

Model	Tf-idf	Acc	Macro avg
MultinomialNB {smoothing=yes}	no	0.6682	0.6661
BernoulliNB {smoothing=yes}	no	0.6878	0.6862
LogisticRegression {multi_class=OVR,max_iter=100}	no	0.7406	0.7387
LinearSVC {multi_class=OVR,loss=squared_hinge}	no	0.7147	0.7120
MultinomialNB {smoothing=yes}	yes	0.6675	0.6626
BernoulliNB {smoothing=yes}	yes	0.6878	0.6862
LogisticRegression {multi_class=OVR,max_iter=100}	yes	0.7295	0.7282
LinearSVC {multi_class=OVR,loss=squared_hinge}	yes	0.7284	0.7265

Figura 4.10: Resultados de varios clasificadores recibiendo el dataset “no-appended-soft-processed“ (preprocesado suave con el “no“ concatenado). Concatenar los “no“ en el preprocesado suave también ofrece un muy leve mejor resultado.

consigue esta técnica es que palabras muy comunes tengan un menor peso y, palabras infrecuentes, consigan un peso mayor. Por ejemplo, la palabra “bien“ se ponderará con un menor peso al ser más común que la palabra “extraordinario“, más infrecuente. Que resulte la ponderación efectiva o no, dependerá de muchísimas variables: la complejidad del lenguaje, las relaciones que los distintos tokens tienen entre sí, el propio dataset y, la forma automática que cada modelo tiene de aprender. Por tanto, prácticamente, estamos obligados a probarlo para saber si nos será útil.

En cuanto a los clasificadores probabilísticos que siguen el teorema de Bayes, podemos ver valores parecidos. Bernoulli Naive Bayes es un método conocido por funcionar especialmente bien con textos cortos. Como se puede comprobar no se ve afectado por el vectorizador Tfidf, pues ignora las frecuencias, trata a todo como booleanos, una palabra puede aparecer o no aparecer. Ambos métodos se ven afectados por el preprocesado, subiendo hasta un 2 % de accurary en el mejor de los casos. Sin embargo, no obtienen una mejora tan grande como los otros. Lo más probable es que se deba a que asumen la hipótesis de independencia entre los tokens, por ello, si una palabra es normalizada o corregida, solo ella misma se “ve afectada“ y no las de su entorno. Aunque estos métodos obtengan peores resultados, cuentan con el aspecto positivo de que son muy rápidos de entrenar.

LogisticRegression y LinearSVC si que consideran dependencia entre los tokens y el preprocesado les afecta en mayor medida. Como se puede observar, aumentan su tasa de accuracy en un 3-4 %. Cabe destacar de estos modelos que son bastante más lentos de entrenar que los otros, sobre todo, LinearSVC.

Como se deduce de los análisis, el mejor tipo de preprocesado para este dataset es el preprocesado suave con los “no“ concatenados delante de las palabras que lo preceden. Sin embargo, las pruebas no son del todo concluyentes como para poder indicar que este tipo de preprocesado es estrictamente mejor que los otros tres con buenos resultados. No obstante, para poder explorar más tipos de clasificadores y vectorizaciones será el que se utilice en adelante. Los resultados nos indican que, aunque la lematización ha dado buenos resultados, no aporta ventajas significativas. Además, cabe

la posibilidad de que el lematizador haya lematizado incorrectamente en ciertas ocasiones, hecho que puede haber generado errores en la clasificación.

4.3.2. Pruebas con otros modelos de Machine Learning tradicionales sobre el mejor dataset preprocesado

En este subapartado se van a analizar distintos tipos de clasificadores, con distintas configuraciones, para encontrar el modelo que mejor se adapte a nuestro problema. Todas las pruebas se realizaron empleando el procesado suave con el "no" concatenado.

La primera prueba que se realizó fue la de utilizar la versión de BinaryMultinomialNB sugerida en [22]. Como se puede observar de los resultados, si que se obtiene una cierta mejora, por lo que si se decidiera emplear este método, resultaría de utilidad.

Model	Tf-idf	Acc	Macro avg
BinaryMultinomialNB {smoothing=yes}	no	0.6735	0.6711

Figura 4.11: Resultados de emplear BinaryMultinomialNB.

Otra prueba que se llevó a cabo fue la de comprobar si empleando bigramas y trigramas se obtendría una mayor tasa de acierto. Dentro del esquema propuesto el usar bigramas implica que los nuevos tokens de los tweets son grupos de dos tokens que se encontraban juntos en la colección [23]. Esto produce un aumento de dimensionalidad. En las figuras 4.12 y 4.13 podemos observar que, de forma general, el acierto ha descendido mucho, siendo peor cuanto mayor es el tamaño del n-gram.

Model	Tf-idf	Acc	Macro avg
MultinomialNB {smoothing=yes}	no	0.6389	0.6353
BernoulliNB {smoothing=yes}	no	0.5887	0.5237
LogisticRegression {multi_class=OVR,max_iter=100}	no	0.6591	0.6562
LinearSVC {multi_class=OVR,loss=squared_hinge}	no	0.6507	0.6472
MultinomialNB {smoothing=yes}	yes	0.6084	0.5955
BernoulliNB {smoothing=yes}	yes	0.5654	0.4812
LogisticRegression {multi_class=OVR,max_iter=100}	yes	0.6264	0.6231
LinearSVC {multi_class=OVR,loss=squared_hinge}	yes	0.6399	0.6380

Figura 4.12: Resultados de emplear 2-grams con varios modelos de clasificación.

Se probaron también modelos basados en redes neuronales. El primero fue un simple Perceptron con el que no se obtuvieron malos resultados, como se puede ver en la figura 4.14, para lo sencillo que es el modelo. La segunda red neuronal que se empleó fue el Perceptron Multicapa y, como se observa en la figura 4.15, no se consiguieron mejores resultados que con la regresión logística o

Model	Tf-idf	Acc	Macro avg
MultinomialNB {smoothing=yes}	no	0.5739	0.5691
BernoulliNB {smoothing=yes}	no	0.4965	0.3841
LogisticRegression {multi_class=OVR,max_iter=100}	no	0.5664	0.5494
LinearSVC {multi_class=OVR,loss=squared_hinge}	no	0.5520	0.5311
MultinomialNB {smoothing=yes}	yes	0.5362	0.5130
BernoulliNB {smoothing=yes}	yes	0.4781	0.3614
LogisticRegression {multi_class=OVR,max_iter=100}	yes	0.5537	0.5361
LinearSVC {multi_class=OVR,loss=squared_hinge}	yes	0.5700	0.5667

Figura 4.13: Resultados de emplear 3-grams con varios modelos de clasificación.

LinearSVC; la precisión se encuentra alrededor del 72 %. Además, para este clasificador no hay una configuración clara. Lo único que se puede afirmar es que la media general de acierto empleando el vectorizador de conteo tfidf para las distintas configuraciones resulta ser mejor. De todas formas, es importante destacar que estas pruebas se realizaron solo empleando validación simple, ya que el coste computacional de entrenar estas redes es altísimo. Las redes más grandes pueden incluso tardar días en entrenar. Por ello, tampoco se puede llegar a conclusiones muy determinantes acerca de las distintas configuraciones.

Model	Tf-idf	Acc	Macro avg
Perceptron	no	0.6973	0.6947
Perceptron	yes	0.6879	0.6872

Figura 4.14: Resultados de emplear el Percetron.

Otro modelo que se utilizó también fue Random Forest, ya que es un modelo que consigue muy buenos resultados en muchos ámbitos. No obtuvo buenos resultados en éste como se puede ver en la figura 4.16. Era algo de esperar porque dimensionalidades tan altas como la de este problema, no resultan buenas para el modelo. A pesar de ello, se puede observar que cuantos más árboles se añadieron, más subió la precisión. Sin embargo, la mejora cada vez era menor y el coste del entrenamiento de este modelo subía rápidamente. Como en las redes neuronales se probó mediante validación simple.

Nuevamente, se probaron máquinas de vectores soporte con otras configuraciones y se obtuvieron ciertas mejoras. Como se puede ver en la figura 4.17 se obtienen muy buenos resultados con SVM tradicionales con un kernel lineal. Hay que destacar que aunque parezcan iguales LinearSVC y las SVM tradicionales con un kernel lineal, son diferentes como se evidencia en [24]. Como se puede apreciar para kernels distintos al lineal consiguieron muy malos resultados. También se probó una variante de las SVM llamada NBSVM que es, en esencia, una SVM con características de Multinomial Naive Bayes. Para más información acudir a [25] y [26]. La precisión curiosamente resultó ser una

Hidden layer sizes	Learning rate	Acc	Macro avg
(300)	constant	0.7163	0.7139
(300)	adaptive	0.7230	0.7201
(150)	constant	0.7204	0.7176
(150)	adaptive	0.7201	0.7182
(100)	constant	0.7187	0.7166
(100)	adaptive	0.7187	0.7153
(50)	constant	0.7200	0.7192
(50)	adaptive	0.7170	0.7138
(500)	adaptive	0.7189	0.7167
(400)	adaptive	0.7219	0.7193
(300,150)	adaptive	0.6999	0.6961

(a) Sin ponderación tf-idf

Hidden layer sizes	Learning rate	Acc	Macro avg
(300)	constant	0.7181	0.7160
(300)	adaptive	0.7234	0.7199
(150)	constant	0.7189	0.7169
(150)	adaptive	0.7117	0.7106
(100)	constant	0.7055	0.7034
(100)	adaptive	0.7147	0.7118
(50)	constant	0.7137	0.7115
(50)	adaptive	0.7050	0.7007
(500)	adaptive	0.7187	0.7169
(400)	adaptive	0.7243	0.7227
(300,150)	adaptive	0.7097	0.7072

(b) Con ponderación tf-idf

Figura 4.15: Resultados para el perceptrón multicapa. Todas las redes utilizan Adam y RELU con un learning inicial de 0.001.

Estimators	Acc	Macro avg
50	0.6529	0.6443
100	0.6632	0.6568
250	0.6701	0.6633
500	0.6726	0.6653
1000	0.6736	0.6664

(a) Sin ponderación tf-idf

Estimators	Acc	Macro avg
50	0.6650	0.6602
100	0.6636	0.6586
250	0.6706	0.6657
500	0.6705	0.6655
1000	0.6739	0.6690

(b) Con ponderación tf-idf

Figura 4.16: Resultados para Random Forest.

mezcla de ambos modelos. Los resultados de las variantes de las SVM se pueden ver en la figura 4.18.

Kernel	Acc	Macro avg
linear	0.7289	0.7267
rbf	0.4291	0.3029
poly	0.3695	0.1799
sigmoid	0.3695	0.1799

(a) Sin ponderación tf-idf

Kernel	Acc	Macro avg
linear	0.7367	0.7361
rbf	0.3695	0.1799
poly	0.3695	0.1799
sigmoid	0.3695	0.1799

(b) Con ponderación tf-idf

Figura 4.17: Resultados para SVM tradicionales.

Model	Tf-idf	Acc	Macro avg
LinearSVC {'multi_class': 'ovr'}	no	0.7146	0.7119
LinearSVC {'multi_class': 'crammer_singer'}	no	0.7143	0.7117
LinearSVC {'multi_class': 'ovr'}	yes	0.7284	0.7265
LinearSVC {'multi_class': 'crammer_singer'}	yes	0.7295	0.7274
NBSVM {'multi_class': 'ovr','c': 1, 'beta': 0.25}	no	0.6968	0.6935
NBSVM {'multi_class': 'ovr','c': 1, 'beta': 0.25}	yes	0.7189	0.7155

Figura 4.18: Resultados de emplear distintas variantes de SVM.

Una alternativa que se decidió probar fue la de emplear diccionarios de palabras positivas y negativas, que se habían extraído de la web [27], para realizar un conteo de cuantas palabras positivas, negativas y desconocidas tenía cada tweet. Aplicando esta transformación al dataset los modelos recibieron tres atributos, uno por cada conteo. Este método puede resultar útil cuando no se tienen suficientes datos para entrenar a los modelos con las técnicas de vectorización mencionadas. Los resultados no fueron muy buenos como se puede ver en la figura 4.19. En general, este acercamiento al problema resulta incorrecto porque muchas palabras, no aparecen en los diccionarios y, por tanto, tienen polaridad desconocida. Los métodos de Machine Learning, recibiendo los tweets vectorizados, siempre que, se tenga un número de datos razonable, conseguirán mejores resultados porque consiguen abstraer mucho mejor la polaridad de las palabras de la colección.

Model	Tf-idf	Acc	Macro avg
MultinomialNB {smoothing=yes}	no	0.5398	0.4914
BernoulliNB {smoothing=yes}	no	0.5310	0.4077
LogisticRegression {multi_class=OVR,max_iter=100}	no	0.5642	0.5476
LinearSVC {multi_class=OVR,loss=squared_hinge}	no	0.5441	0.5165

Figura 4.19: Resultados de emplear diccionarios de polaridad

El mejor modelo hasta el momento seguía siendo la regresión logística y, por ello, se hicieron pruebas para encontrar su mejor configuración de hiperparámetros. Como se puede ver en la figura 4.20, se encontró una configuración con la que se obtuvo el mejor valor de precisión en entrenamiento hasta el momento 74.11 %. Esta configuración emplea la estrategia multiclase multinomial en vez de one-vs-rest y, también utiliza como solver, saga. Saga es muy apropiado para nuestro problema ya que es el mejor para sparse multinomial logistic regression como se menciona en [28].

Multi class	Solver	Max iters	Acc	Macro avg
OVR	liblinear	100	0.7406	0.7387
multinomial	newton-cg	100	0.7374	0.7353
multinomial	lbfgs	100	0.7383	0.7364
multinomial	sag	100	0.7389	0.7369
multinomial	saga	100	0.7411	0.7391
multinomial	saga	500	0.7376	0.7355
multinomial	saga	1000	0.7375	0.7354
multinomial	saga	2000	0.7375	0.7354

(a) Sin ponderación tf-idf

Multi class	Solver	Max iters	Acc	Macro avg
OVR	liblinear	100	0.7295	0.7282
multinomial	newton-cg	100	0.7318	0.7305
multinomial	lbfgs	100	0.7323	0.7310
multinomial	sag	100	0.7318	0.7305
multinomial	saga	100	0.7318	0.7305
multinomial	saga	500	0.7318	0.7305
multinomial	saga	1000	0.7318	0.7305
multinomial	saga	2000	0.7318	0.7305

(b) Con ponderación tf-idf

Figura 4.20: Resultados de emplear regresión logística

Las últimas pruebas que se llevaron a cabo fueron entrenar una regresión logística que recibía como datos los dense vectors generados mediante la técnica Doc2Vec. Los resultados no fueron buenos como se puede observar en la figura 4.21. A su vez, como se evidencia, este tipo de representación no resulta efectiva. El motivo principal es que la técnica empleada agrupa en el espacio vectorial los documentos no solo por su polaridad, sino también por otros aspectos como podrían ser su estructura interna. Por ejemplo, los tweets de política suelen tener un formato más formal, los de entretenimiento uno más distendido, pero ambos contienen tweets de las tres clases.

Negative sampling	Vector size	Acc	Macro avg
5	500	0.5938	0.5912
5	1000	0.5988	0.5950
5	2000	0.5919	0.5886
11	500	0.6060	0.6034
11	1000	0.6028	0.5997
11	1000	0.6048	0.6016

(a) Empleando el algoritmo PV-DM

Negative sampling	Vector size	Acc	Macro avg
5	500	0.6400	0.6379
5	1000	0.6383	0.6362
5	2000	0.6437	0.6417
11	500	0.6388	0.6365
11	1000	0.6442	0.6421
11	2000	0.6422	0.6396

(b) Empleado el algoritmo PV-DBOW

Figura 4.21: Resultados de emplear regresión logística utilizando dense vectors que representan a cada documento mediante al algoritmo Doc2Vec.

4.3.3. Pruebas con modelos de deep learning

Los modelos de deep learning que se emplearon fueron LSTMs y GRUs, ambas con y sin bidireccionalidad. En un primer lugar, se probaron añadiendo a cada uno de estos modelos una capa de embedding. Esta capa permite que las redes aprendan los vectores de cada palabra por sí mismas, el tamaño de cada uno de los vectores se fijo en 128. A ambos modelos se les añadió también dropout al 40 % que apaga para cada documento de entrada ciertas neuronas para así prevenir el overfitting. La implementación de los modelos se puede ver con detalle en el apéndice B.1. Los resultados conseguidos fueron muy buenos superando a la regresión logística, como se puede ver en la figura 4.22. Este tipo de modelos recurrentes son capaces de aprender dependencias entre distintos tokens de los documentos, aunque se encuentren distanciados. Por este motivo consiguen tan buenos resultados. El mejor modelo es la red GRU con 200 unidades sin bidireccionalidad con un 74.74 % de acierto en test y la red LSTM con 100 unidades sin bidireccionalidad con un 74.34 %. Como se puede observar, la bidireccionalidad no ha resultado ser un factor determinante. Todas las redes convergieron a la mejor solución en la segunda época. En las épocas sucesivas, la red se encontraba sobreaprendida y reducía su precisión en test como se puede ver en la figura 4.23.

Cell units	Bidirectional	Acc
200	no	0.7432
200	yes	0.7428
100	no	0.7434
100	yes	0.7391

(a) LSTM

Cell units	Bidirectional	Acc
200	no	0.7474
200	yes	0.7392
100	no	0.7396
100	yes	0.7433

(b) GRU

Figura 4.22: Resultados de emplear LSTMs y GRUs. Ambos modelos contaban con una capa de Embedding y una capa de Dropout al 40 %.

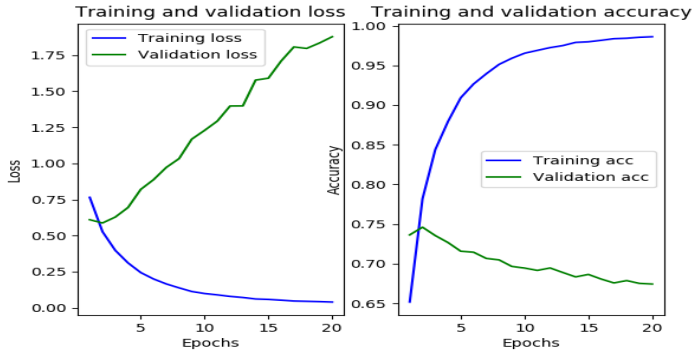


Figura 4.23: Muestra del overfitting para una red LSTM.

Una vez probada la capa de auto embedding, se decidió usar la técnica de Word2Vec para construir de forma externa a la red los vectores de cada token. Ya construidos estos vectores, de 128 de

dimensión, se entrenaron varios de los modelos anteriores. Los resultados obtenidos no fueron muy buenos como se puede apreciar en la imagen 4.24. Además, estas redes no sólo tenían una precisión peor, sino que tardaban más épocas en aprender e, igualmente, presentaban overfitting. Salvo una de las redes anteriores, que llegó al máximo de iteraciones (10), tres de los modelos se pararon con early stopping, sobre la quinta época, porque eran incapaces de mejorar la tasa de acierto en test. Se deduce de estos resultados que en este caso resulta fundamental que las redes puedan aprender sus propios word embeddings.

Model	Cell units	Bidirectional	Acc
LSTM	200	no	0.6686
GRU	200	yes	0.6699
LSTM	200	no	0.6692
GRU	200	yes	0.6732

Algorithm	Vector size	Window size	Negative sampling	Alpha
Skip-gram	128	2	20	0.0300

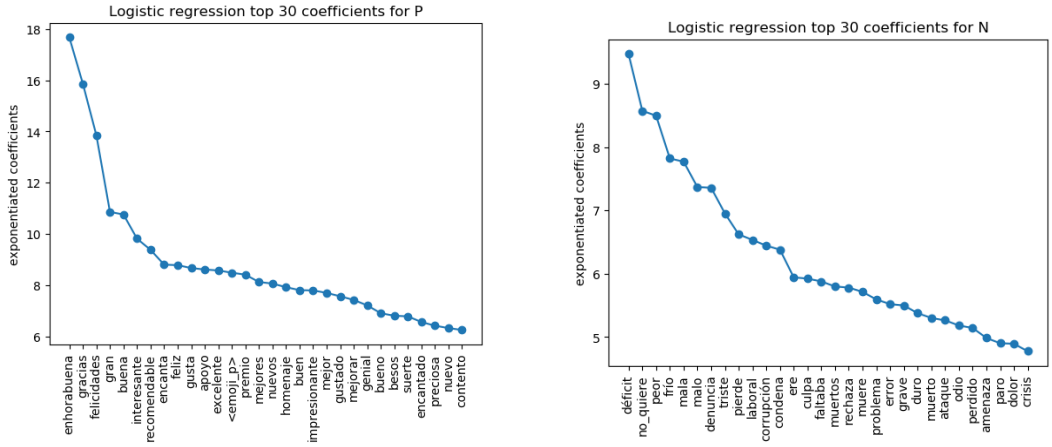
Figura 4.24: Resultados de emplear LSTMs y GRUs con vectores densos generados de forma externa a las redes con Word2Vec.

4.3.4. Interpretación de los dos mejores modelos

Se decidió analizar dos de los mejores modelos obtenidos, la red GRU y la regresión logística, para comprobar de qué forma estos métodos habían resuelto el problema.

Para la regresión logística se realizó un análisis de sus pesos. Como se puede observar de la figura 4.25, para los pesos de la clase positiva está aportando un valor muy grande en el coeficiente de aquellas palabras muy positivas y muy frecuentes. Lo mismo ocurre para el vector de pesos de la clase negativa. Esto confirma la afirmación dicha anteriormente de que las palabras más frecuentes y con una polaridad más marcada son muy determinantes y, es por ello, que el modelo les da una mayor importancia. Se puede ver en la figura 4.26 que varias de las palabras que aparecen en los pesos se corresponden a las palabras más frecuentes y más cohesionadas con la clase positiva y negativa del dataset.

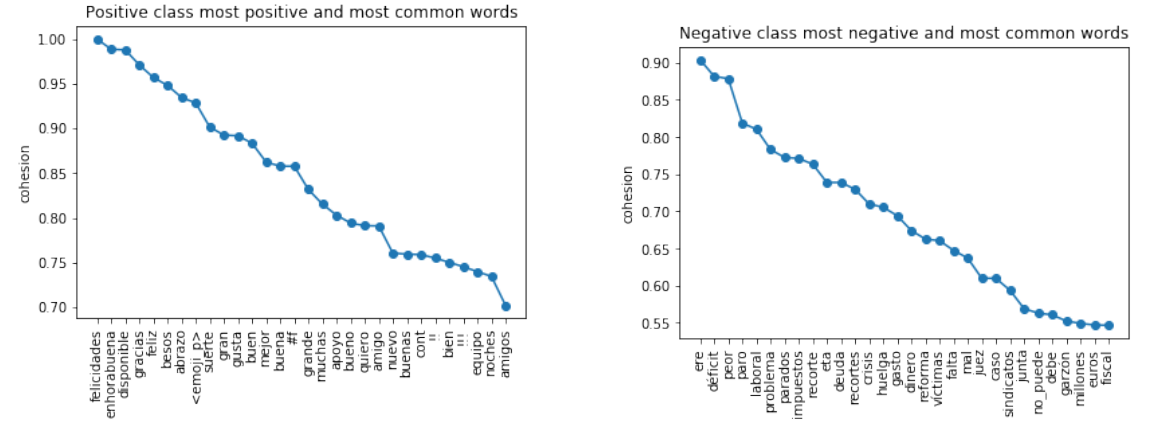
Para la red GRU se extraen 900 word embeddings, compuestos por los 300 tokens más frecuentes de cada clase. Realizando la clusterización, explicada en el apartado de desarrollo, se puede observar en la figura 4.27 que los tres grupos del clustering aparentemente se encuentran organizados por polaridad, el primero para la clase negativa, el segundo para la neutra y, el tercero, para la positiva. Esto demuestra que la capa de Embedding es una herramienta esencial que la red tiene para poder ajustar la polaridad de las palabras, ya que, como se mostró anteriormente, si no se le permite a la red generar sus propios embeddings no obtiene buenos resultados.



(a) Valor de los coeficientes del vector positivo

(b) Valor de los coeficientes del vector negativo

Figura 4.25: Valores de los coeficientes de la regresión logística y la palabra asociada.



(a) Palabras más positivas y más frecuentes de la clase positiva

(b) Palabras más negativas y más frecuentes de la clase negativa

Figura 4.26: Se muestran de las 100 palabras más frecuentes aquellas 30 que están más cohesionadas con su clase según la métrica WCCo descrita anteriormente.

Grupo 1	recortes	ni	menos	crisis	puede	millones	laboral	nada	parece	déficit
Grupo 2	<nick>	<url>	partido	hoy	día	¿	madrid	congreso	política	mañana
Grupo 3	rt	<emoji_p>	gracias	buenos	gran	<jaja>	mejor	nuevo	enhorabuena	reforma

Figura 4.27: Muestra de 10 palabras, elegidas por su alta frecuencia en el dataset, para cada uno de los grupos del clustering realizado a los word embeddings de la red GRU.

4.3.5. Resultados finales en el conjunto de validación

Por último, se probaron los tres mejores modelos en el conjunto de validación que todavía no había sido visto por ningún modelo. La red GRU con auto-embeddings obtuvo un 75.42 %, la red LSTM con auto-embeddings un 75.40 % y la regresión logística multinomial 75.09 %. En las curvas ROC de las figuras 4.28 y 4.29 se pueden ver representadas las curvas ROC de cada clase contra el resto. De ella se puede observar que la clase mejor clasificada ha sido la positiva, la segunda, la negativa y, la tercera, la neutra.

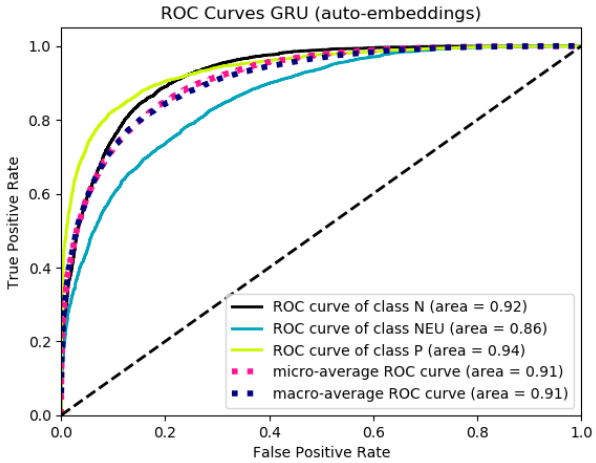
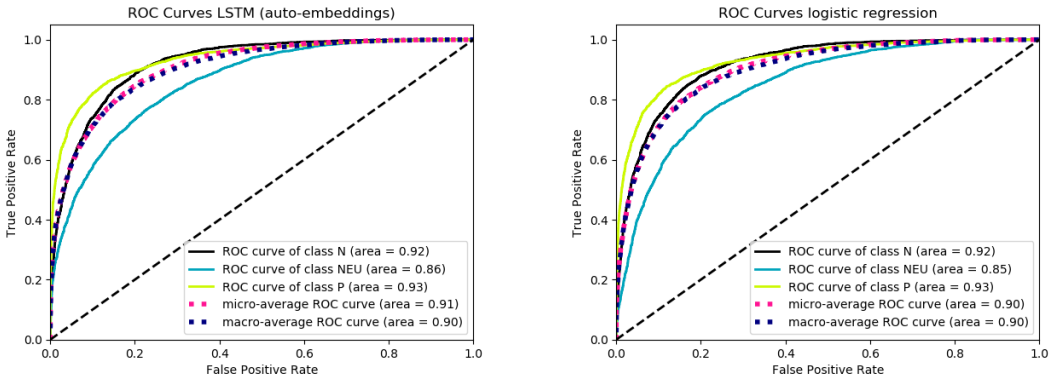


Figura 4.28: Curvas ROC para la GRU



(a) Curvas ROC para la LSTM

(b) Curvas ROC para la regresión logística multinomial

Figura 4.29: Curvas ROC para la LSTM y para la regresión logística

5 Conclusiones y trabajo futuro

5.1. Conclusiones

En este proyecto se ha llevado a cabo una tarea de Sentiment Analysis de tweets escritos en español empleando algoritmos de Machine Learning tradicionales y deep learning. Durante su desarrollo se pudo comprobar que había bastante carencia de contenido para el análisis en español. Este contenido incluye datasets en los que, de forma pública, apenas hay disponibles datos fiables. El utilizado en este proyecto, aparte de poder ser utilizado sólo para el ámbito de investigación, es del año 2012, cuando los tweets eran la mitad de cortos. También, hay escasez de herramientas para tratamiento del lenguaje en español, como pueden ser lematizadores, analizadores sintácticos e instrumentos útiles para Sentiment Analysis, como los diccionarios de polaridad. La metodología de trabajo ha sido la tradicional en este tipo de proyectos, que consiste en un preprocesado, un modelado y una clasificación de los documentos.

La conclusión extraída de la aplicación de preprocesado de los tweets es que, de forma general, ha resultado bastante beneficiosa. Con ambos tipos de preprocesados, los denominados como suave y fuerte, se ha conseguido un aumento de precisión, mayor del 2 %. Por tanto, si se desea en este tipo de proyectos aumentar la precisión lo máximo posible resultan fundamentales.

En cuanto al modelado del lenguaje la utilización de sparse vectors para algoritmos tradicionales ha ofrecido buenos resultados. En contraste, la generación manual de dense vectors, utilizando algoritmos como Word2vec o Doc2Vec, no ha dado buenos resultados para el tamaño de nuestro dataset. Sin embargo, las capas de Embedding de las redes profundas si que consiguen generar correctamente word embeddings.

Las técnicas de aprendizaje automático que obtuvieron los mejores resultados fueron las redes profundas, en concreto, las GRUs y las LSTMs. De las técnicas tradicionales, las que resultaron obtener los mejores resultados, fueron la regresión logística y máquinas de vectores soporte, empleando sparse vectors, algo no sorprendente porque éstas técnicas consiguen buenos resultados de forma consistente en proyectos de NLP.

De la interpretación realizada sobre la regresión logística se pudo comprobar la intuición de que, como estos modelos solo pueden aprender realizando un ajuste de las polaridades de las palabras, ponderaban de forma muy alta aquellas palabras muy frecuentes con una gran cohesión a su clase. El análisis del mejor modelo, la red GRU, indicó que la capa de Embedding permite al modelo vectorizar las palabras agrupándolas en el espacio, en función de su polaridad.

Finalmente, mencionar que mejor el acierto total conseguido ha sido de aproximadamente un 75 % en el set de validación, con las redes profundas ya mencionadas, siendo las clases más fáciles de clasificar la positiva, la negativa y, en último lugar, la neutra.

5.2. Trabajo futuro

Extensiones de este proyecto interesantes de implementar incluyen alternativas de deep learning empleando redes convolucionales, dividiendo los documentos, tanto a nivel de token, como de caracter, con las técnicas explicadas en [29]. Otro modelo a probar sería la versión supervisada de FastText con la que se podría realizar una comparación entre los resultados obtenidos y el tiempo de entrenamiento entre este modelo y los paradigmas de deep learning. También, resultaría bastante relevante repetir las pruebas realizadas de este proyecto en otros datasets para ver si los resultados obtenidos son una tendencia sistemática para este tipo de proyectos de Sentiment Analysis.

Distinto al ámbito de la clasificación, resultaría curioso llevar a cabo un análisis de la visualización de los “firings“ de las neuronas de los modelos recurrentes. Con ello se examinaría el comportamiento interno de la red y, quizás, se podría observar que ciertas neuronas se activan para tokens de determinada polaridad. Este tipo de visualización se expone en el siguiente artículo [30]. Por último, una propuesta, bastante de actualidad, sería comprobar si el clasificador ha desarrollado ciertos sesgos positivos o negativos hacia algún género o etnia en particular.

Bibliografía

[1] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[2] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, 2017.

[3] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," *arXiv preprint arXiv:1607.01759*, 2016.

[4] J. Villena-Román, S. Lana-Serrano, E. Martínez-Cámara, and J. C. González-Cristóbal, "Tass - workshop on sentiment analysis at sepln. procesamiento del lenguaje natural, 50.," *http://journal.sepln.org*, 2013.

[5] I. Mozetič, M. Grčar, and J. Smailović., "Multilingual twitter sentiment classification: The role of human annotators," *PloS one*, 2016.

[6] D. Jurafsky, *Speech and Language Processing*, p. 70. third edition draft ed., 2018.

[7] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky., "The stanford corenlp natural language processing toolkit in proceedings of the 52nd annual meeting of the association for computational linguistics: System demonstrations," *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, 2014.

[8] "Diccionario de lemas." https://github.com/michmech/lemmatization-lists/blob/master/lemmatization-es.txt. Accessed: 19-5-19.

[9] "Nltk 3.4.1 documentation." https://www.nltk.org/. Accessed: 19-5-19.

[10] "Scikit vectorizers." https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text. Accessed: 19-5-19.

[11] "Scikit pipeline." https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html. Accessed: 19-5-19.

[12] "Gensim word2vec word embeddings." https://radimrehurek.com/gensim/models/word2vec.html. Accessed: 12-5-19.

[13] "Gensim doc paragraph embeddings." https://radimrehurek.com/gensim/models/doc2vec.html. Accessed: 19-5-19.

[14] O. Levy and Y. Goldberg, "Dependency-based word embeddings," 2014.

[15] "Negative sampling in word2vec." https://stats.stackexchange.com/questions/244616/how-does-negative-sampling-work-in-word2vec. Accessed: 19-5-19.

[16] "Scikit documentation v0.19.2." https://scikit-learn.org/0.19/documentation.html. Accessed: 19-5-19.

[17] "Keras recurrent layers." https://keras.io/layers/recurrent. Accessed: 19-5-19.

[18] "Keras callbacks." https://keras.io/callbacks/. Accessed: 19-5-19.

[19] “Ley de zipf.” https://es.wikipedia.org/wiki/Ley_de_Zipf. Accessed: 19-5-19.

[20] “Scikit agglomerative clustering.” https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html. Accessed: 19-5-19.

[21] “Model evaluation.” https://scikit-learn.org/stable/modules/model_evaluation.html. Accessed: 19-5-19.

[22] D. Jurafsky, *Speech and Language Processing*, pp. 70–72. third edition draft ed., 2018.

[23] “N-grama.” https://es.wikipedia.org/wiki/N-grama. Accessed: 19-5-19.

[24] “Difference between svc and linearsvc.” https://datascience.stackexchange.com/questions/6987/can-you-explain-the-difference-between-svc-and-linearsvc-in-scikit-learn. Accessed: 19-5-19.

[25] S. Wang and C. D. Manning, “Baselines and bigrams: Simple, good sentiment and topic classification,” in *Proceedings of the 50th annual meeting of the association for computational linguistics: Short papers-volume 2*, pp. 90–94, Association for Computational Linguistics, 2012.

[26] “Implementación nbsvm.” https://github.com/prakhar-agarwal/Naive-Bayes-SVM. Accessed: 19-5-19.

[27] “Sentiment lexicons for 81 languages.” https://www.kaggle.com/rtatman/sentiment-lexicons-for-81-languages. Accessed: 19-5-19.

[28] “Scikit solvers.” https://scikit-learn.org/stable/modules/linear_model.html. Accessed: 19-5-19.

[29] “Understanding convolutional neural networks for nlp.” http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/. Accessed: 19-5-19.

[30] A. Karpathy, J. Johnson, and L. Fei-Fei, “Visualizing and understanding recurrent networks,” *arXiv preprint arXiv:1506.02078*, 2015.

Apéndices

Apéndice A

A.1. Diagrama de clases del preprocesado

En la figura A.1 se muestra el diagrama de clases para los módulos que se han explicado en la sección 3.2. El módulo principal es “Preprocessor“ que hace uso toda la funcionalidad de los otros módulos.

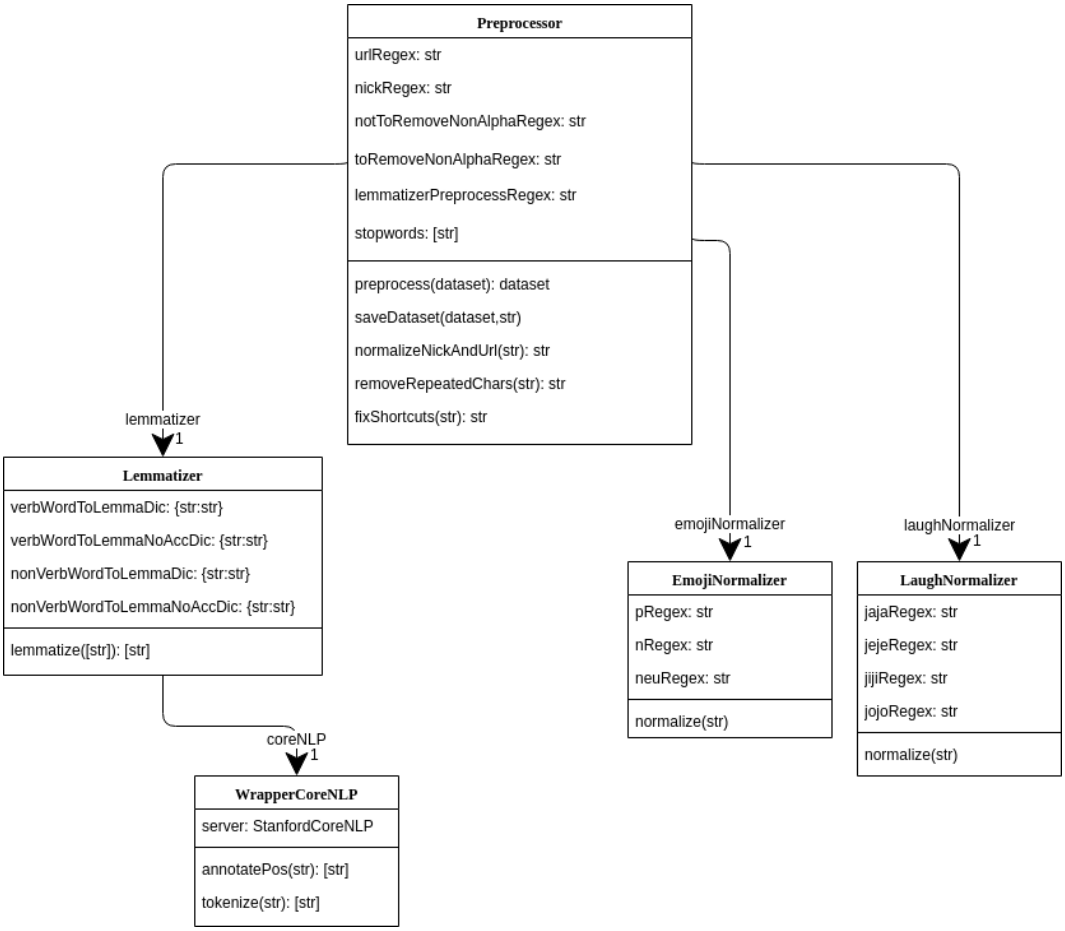


Figura A.1: Este diagrama de clases muestra la relación entre los módulos desarrollados para el preprocesado.

B

Apéndice B

B.1. Creación de redes profundas

En los siguientes fragmentos de código se puede ver paso por paso el ajuste a los datos, la creación del modelo y el entrenamiento de una red GRU empleando la librería Keras. En la figura B.1 se muestra un resumen de la arquitectura de la red descrita en el código.

Código B.1: En este fragmento de código se incluyen las librerías que serán necesarias y las modificaciones necesarias que hay que aplicarle a los datos.

```
1  import numpy as np
2  import pandas as pd
3
4  import tensorflow as tf
5
6  from sklearn.model_selection import train_test_split
7
8  dataset = ... # dataset empleado para train/test
9
10 INPUT_LENGTH = 35
11 EMBED_DIM = 128
12
13 # Calculando el número total de tokens
14 tokens = set()
15 for words in dataset.data:
16     for w in words:
17         tokens.add(w)
18 dim = len(tokens)
19
20 # Uso del tokenizador
21 tokenizer = tf.keras.preprocessing.text.Tokenizer(num_words=dim,
22     filters='',lower=False,split='␣')
23 # nuestro dataset ya esta tokenizado y por ello es necesario este hack
24 tokenizer.fit_on_texts(["␣".join(x) for x in dataset.data])
25 sequences = tokenizer.texts_to_sequences(["␣".join(x) for x in dataset.data])
26 X_values = tf.keras.preprocessing.sequence.pad_sequences(sequences,
27     maxlen=INPUT_LENGTH)
28 labels = pd.get_dummies(dataset.target).values
```

Código B.2: En este fragmento de código se incluye la creación del modelo y el entrenamiento del mismo.

```
DROPOUT_VAL = 0.4
BATCH_SIZE = 32
UNITS = 200
DENSE_DIM = 3
TRAIN_EPOCHS = 10
MODEL_PATH = 'example.h5'

# Construcción del modelo
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Embedding(dim, EMBED_DIM,input_length=INPUT_LENGTH))
model.add(tf.keras.layers.SpatialDropout1D(DROPOUT_VAL))
model.add(tf.keras.layers.GRU(UNITS, dropout=DROPOUT_VAL,
    recurrent_dropout=DROPOUT_VAL))
model.add(tf.keras.layers.Dense(DENSE_DIM,activation='softmax'))
model.compile(loss = 'categorical_crossentropy', optimizer='adam',
    metrics = ['accuracy'])
print(model.summary())

# Fast stopping y checkpoint para guardar el mejor modelo
es = tf.keras.callbacks.EarlyStopping(monitor='val_loss', mode='min',
    verbose=0, patience=3)
mc = tf.keras.callbacks.ModelCheckpoint(MODEL_PATH, monitor='val_acc',
    mode='max', verbose=0, save_best_only=True)

# Entrenamiento
model.fit(X_train, y_train, epochs=TRAIN_EPOCHS, validation_data=(X_test, y_test),
    batch_size=BATCH_SIZE, verbose=1, callbacks=[es,mc])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 35, 128)           6289152
_________________________________________________________________
spatial_dropout1d (SpatialDr (None, 35, 128)           0
_________________________________________________________________
gru (GRU)                    (None, 200)               197400
_________________________________________________________________
dense (Dense)                (None, 3)                 603
=================================================================
Total params: 6,487,155
Trainable params: 6,487,155
Non-trainable params: 0
_________________________________________________________________
```

Figura B.1: Resumen de la arquitectura de la red mostrada por Keras para la creación del modelo ofrecido en el código anterior.


UAM
UNIVERSIDAD AUTONOMA
DE MADRID